

Report S2

Kengo Nakajima

Technical & Scientific Computing II (4820-1028)

Seminar on Computer Science II (4810-1205)

Parallel FEM

- Overview
- Distributed Local Data
- Program
- Results

1D Static Linear Elastic Problem



- Only deforms in x -direction (displacement: u)
 - Uniform: Sectional Area A , Young's Modulus E
 - Boundary Conditions (B.C.)
 - $x=0$: $u=0$ (fixed)
 - $x=x_{max}$: F (axial force)
- Truss: NO bending deformation by G-force

Copy and Compile

Copy

```
>$ cd <$0-fem2>
```

```
>$ cp /home/z30088/fem2/s2r.tar .
```

```
>$ tar xvf s2r.tar
```

<\$0-fem2>/mpi/S2-ref is created

<\$0-fem2>/mpi/S2-ref is called as **<\$0-S2r>**

Compile

```
>$ cd <$0-S2r>
```

```
>$ mpifccpx -Kfast 1d.c
```

Control File: input.dat

Control Data input.dat

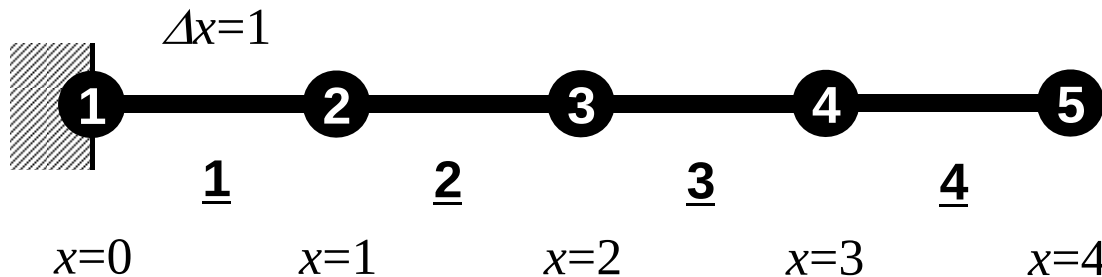
4
1.0 1.0 1.0 1.0
100
1.e-8

NE (Number of Elements) (=40)
 Δx (Length of Each Elem.: L) , F, A, E
Number of MAX. Iterations for CG Solver
Convergence Criteria for CG Solver

$$\sigma = \frac{F}{A} = \frac{1}{1} = 1$$

$$\frac{du}{dx} = \varepsilon = \frac{\sigma}{E} = \frac{1}{1} = 1$$

Strain= 1= 100%



Element ID
Node ID (Global)

go.sh

```
#!/bin/sh
#PJM -L "node=4"           Node # (.1e.12)
#PJM -L "elapsed=00:10:00" Comp.Time (.1e.15min)
#PJM -L "rscgrp=lecture4"  "Queue" (or lecture)
#PJM -g "gt74"            "Wallet"
#PJM -
#PJM -o "test.lst"        Standard Output
#PJM --mpi "proc=64"      MPI Process # (.1e.192)

mpiexec ./a.out
```

N=8
"node=1"
"proc=8"

N=16
"node=1"
"proc=16"

N=32
"node=2"
"proc=32"

N=64
"node=4"
"proc=64"

N=192
"node=12"
"proc=192"

Procedures for Parallel FEM

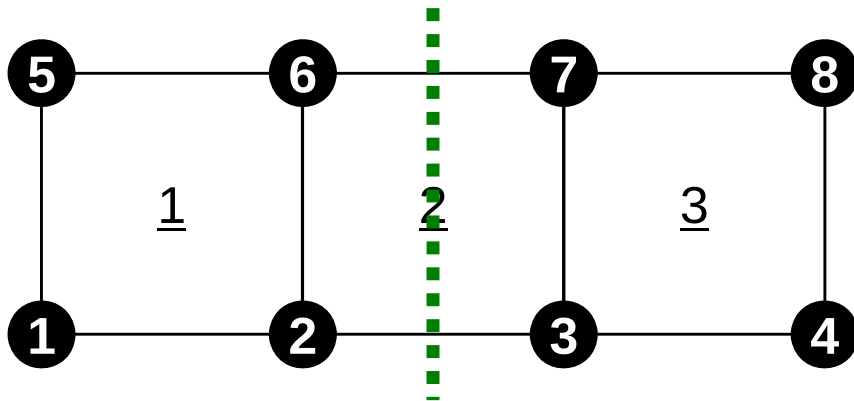
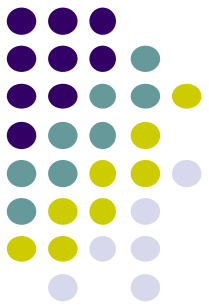
- Reading control file, entire element number etc.
- Creating “distributed local data” in the program
- Assembling local and global matrices for linear solvers
- Solving linear equations by CG
- Not so different from those of original code

- Overview
- **Distributed Local Data**
- Program
- Results

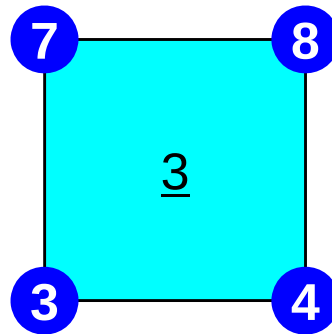
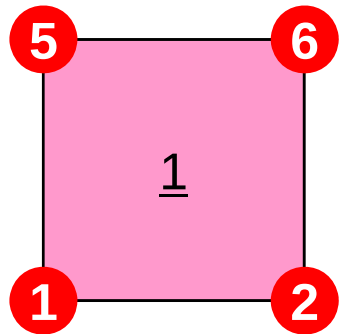
Finite Element Procedures

- Initialization
 - Control Data
 - Node, Connectivity of Elements (N: Node#, NE: Elem#)
 - Initialization of Arrays (Global/Element Matrices)
 - Element-Global Matrix Mapping (Index, Item)
- Generation of Matrix
 - Element-by-Element Operations (do icel= 1, NE)
 - Element matrices
 - Accumulation to global matrix
 - Boundary Conditions
- Linear Solver
 - Conjugate Gradient Method
- Calculation of Stress

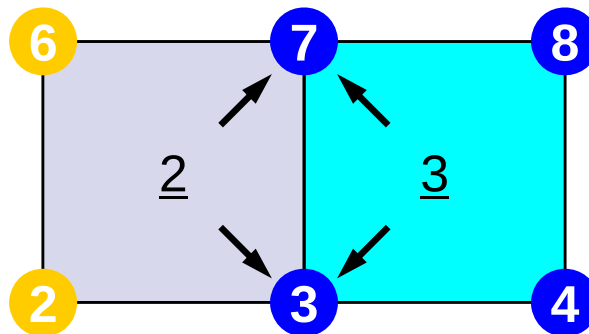
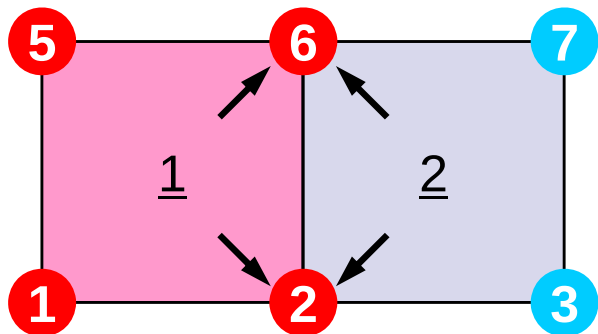
Quadrilateral Elements



Node-based partitioning
Independent variables are defined at nodes (vertices).

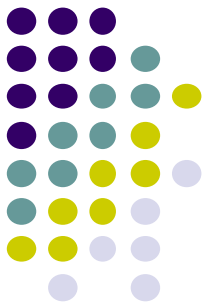


Information is not sufficient for assembling coefficient matrices.



Info. of nodes and elements in overlapped zones are required for assembling coef. matrices.
Computation of stress components needs same info.

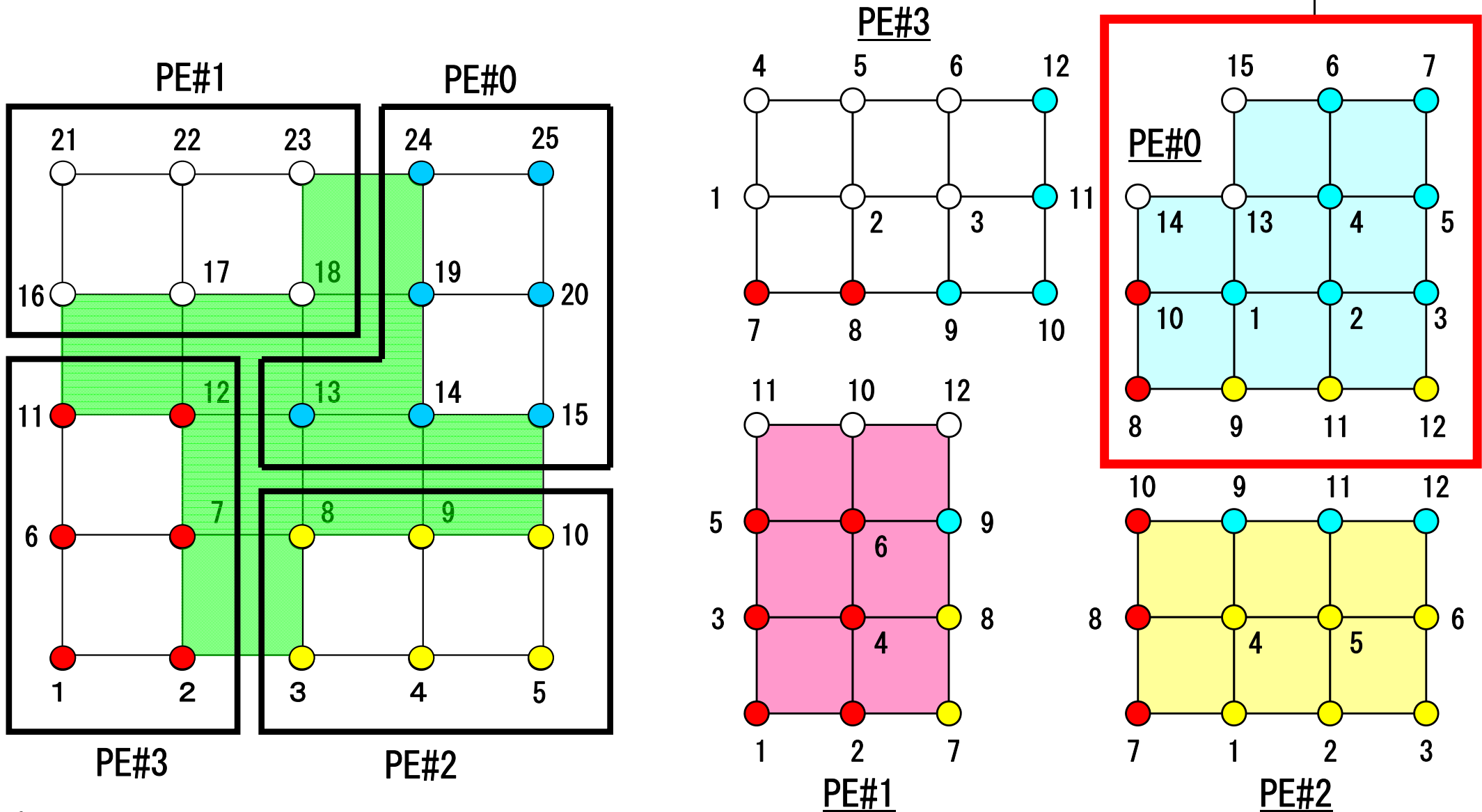
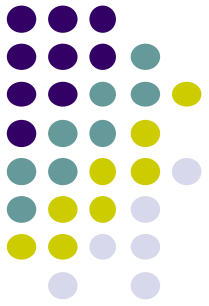
Distributed Local Data Structure for Parallel FEM



- **Node-based partitioning**
- Local data includes:
 - Nodes originally assigned to the domain/PE/partition
 - Elements which include above nodes
 - Nodes which are included above elements, and originally NOT-assigned to the domain/PE/partition
- 3 categories for nodes
 - **Internal nodes** Nodes originally assigned to the domain/PE/partition
 - **External nodes** Nodes originally NOT-assigned to the domain/PE/partition
 - **Boundary nodes** External nodes of other domains/PE's/partitions
- Communication tables
- Global info. is not needed except relationship between domains
 - Property of FEM: local element-by-element operations

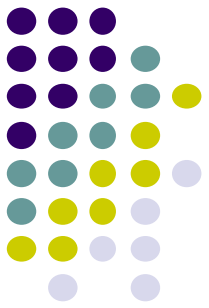
Node-based Partitioning

internal nodes - elements - external nodes

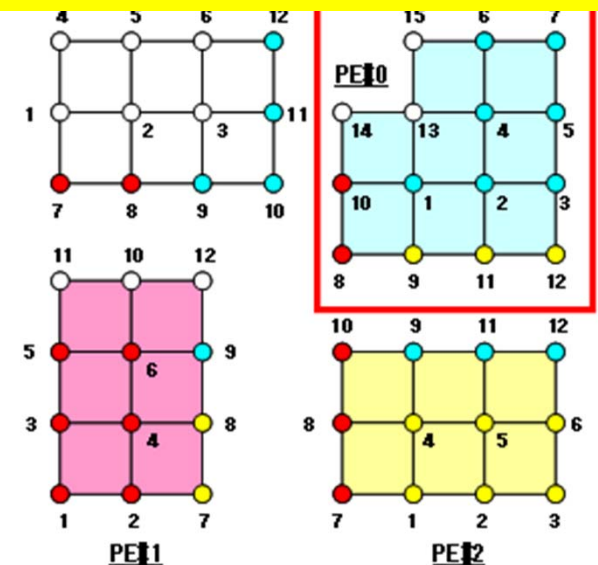
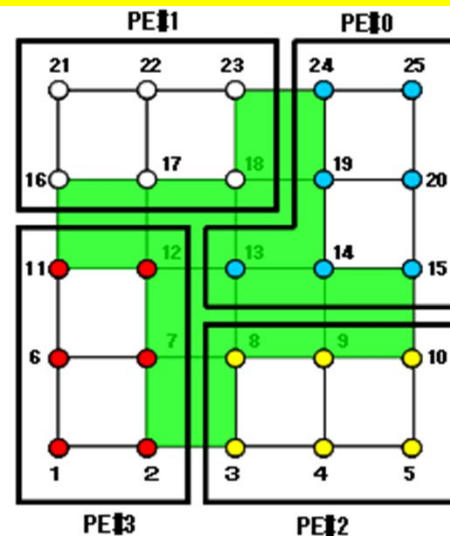
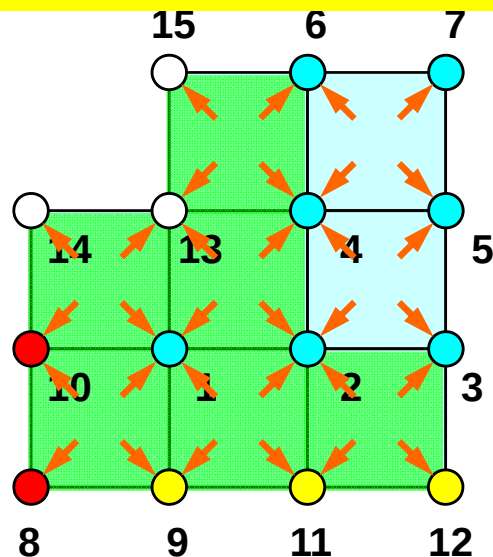


Node-based Partitioning

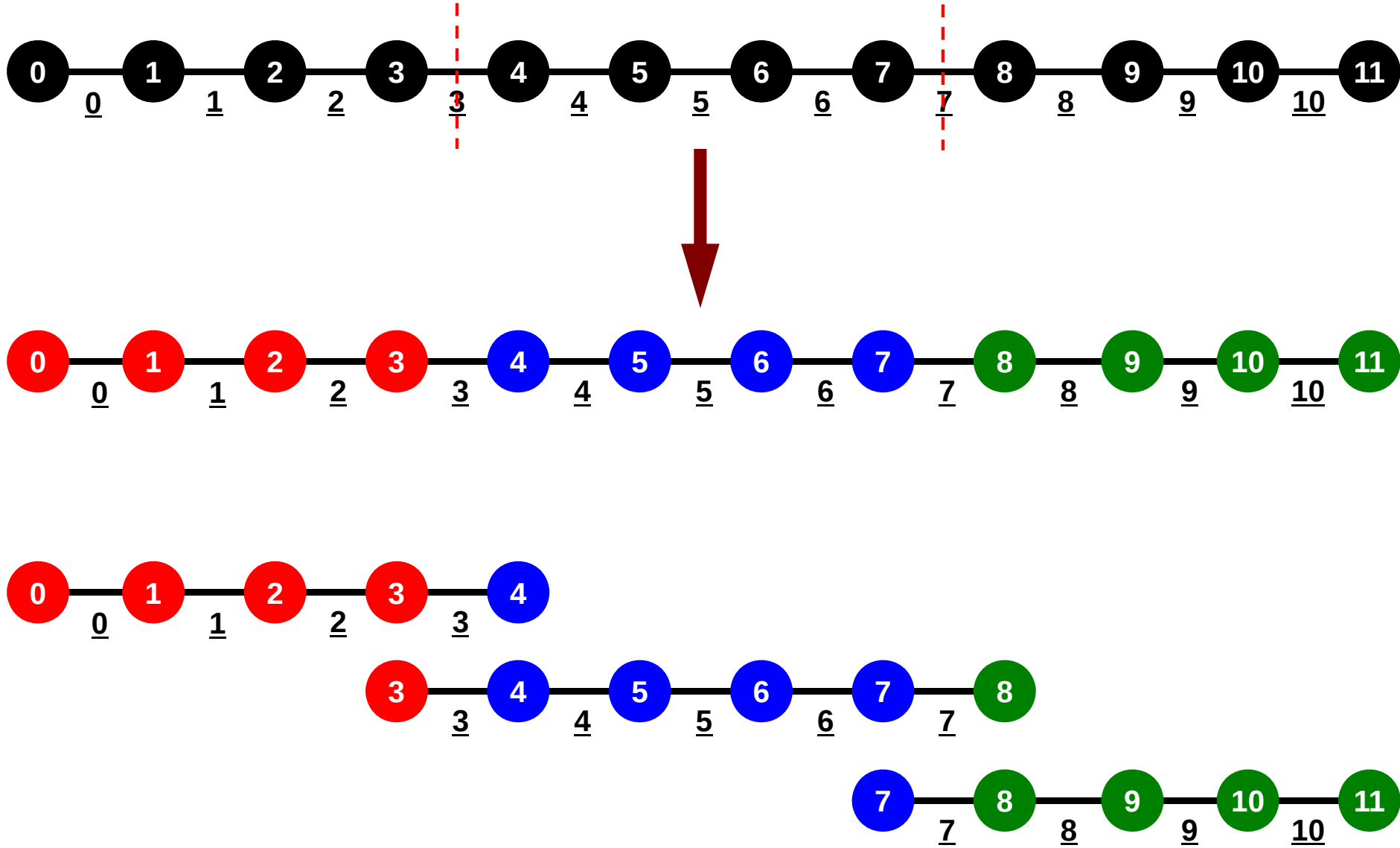
internal nodes - elements - external nodes



- Partitioned nodes themselves (Internal Nodes)
- Elements which include Internal Nodes
- External Nodes included in the Elements
in overlapped region among partitions.
- Info of External Nodes are required for completely local
element-based operations on each processor.

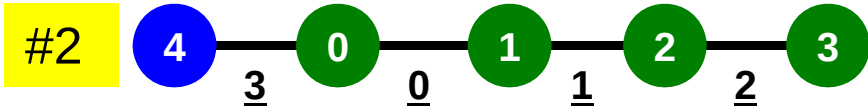
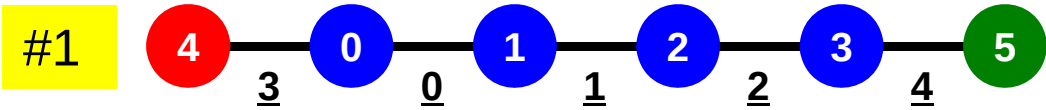
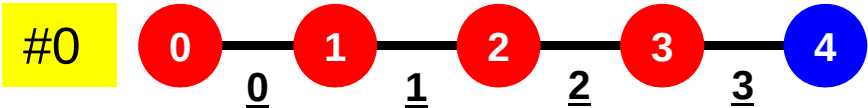


1D FEM: 12 nodes/11 elem's/3 domains



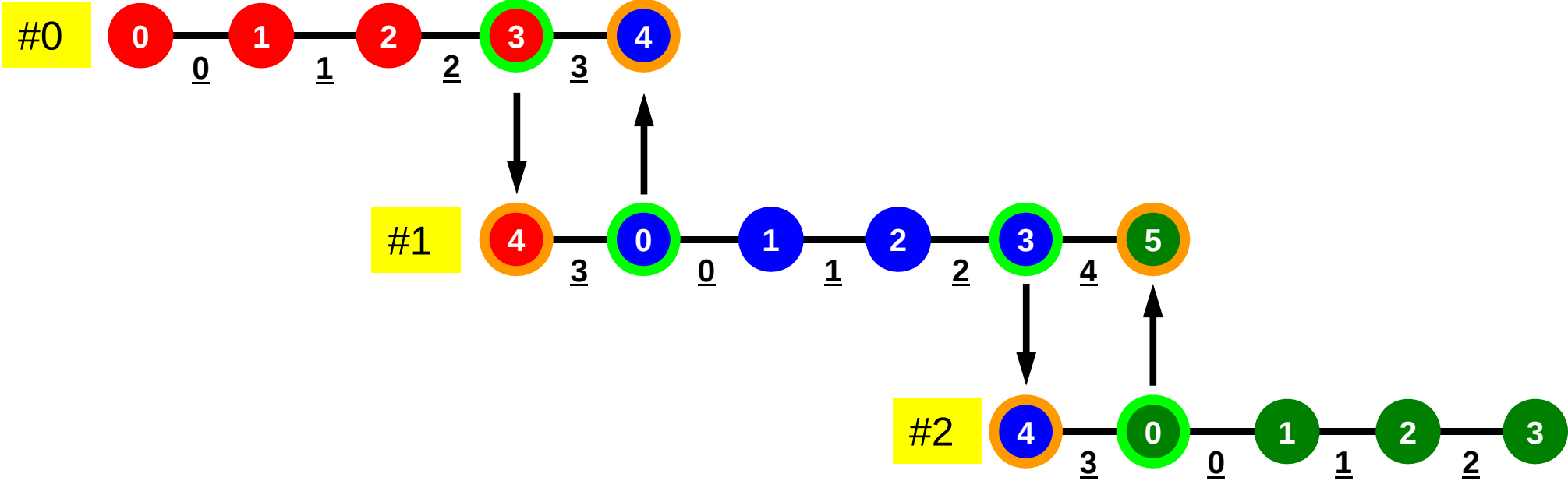
1D FEM: 12 nodes/11 elem's/3 domains

Local ID: Starting from 0 for node and elem at each domain

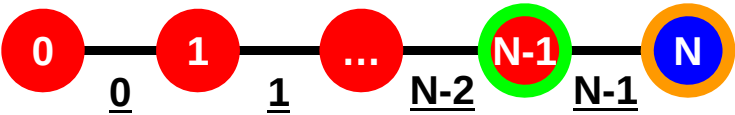


1D FEM: 12 nodes/11 elem's/3 domains

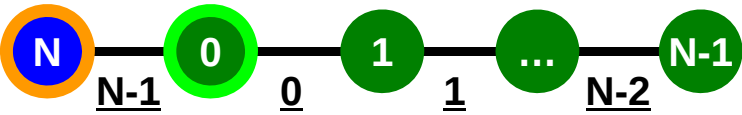
Internal/External Nodes



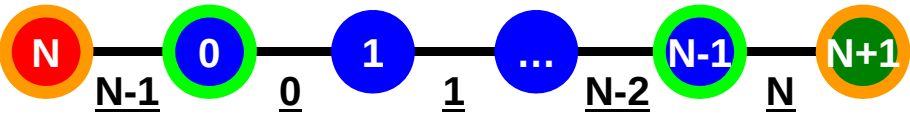
1D FEM: Numbering of Local ID



#0:
N+1 nodes
N elements



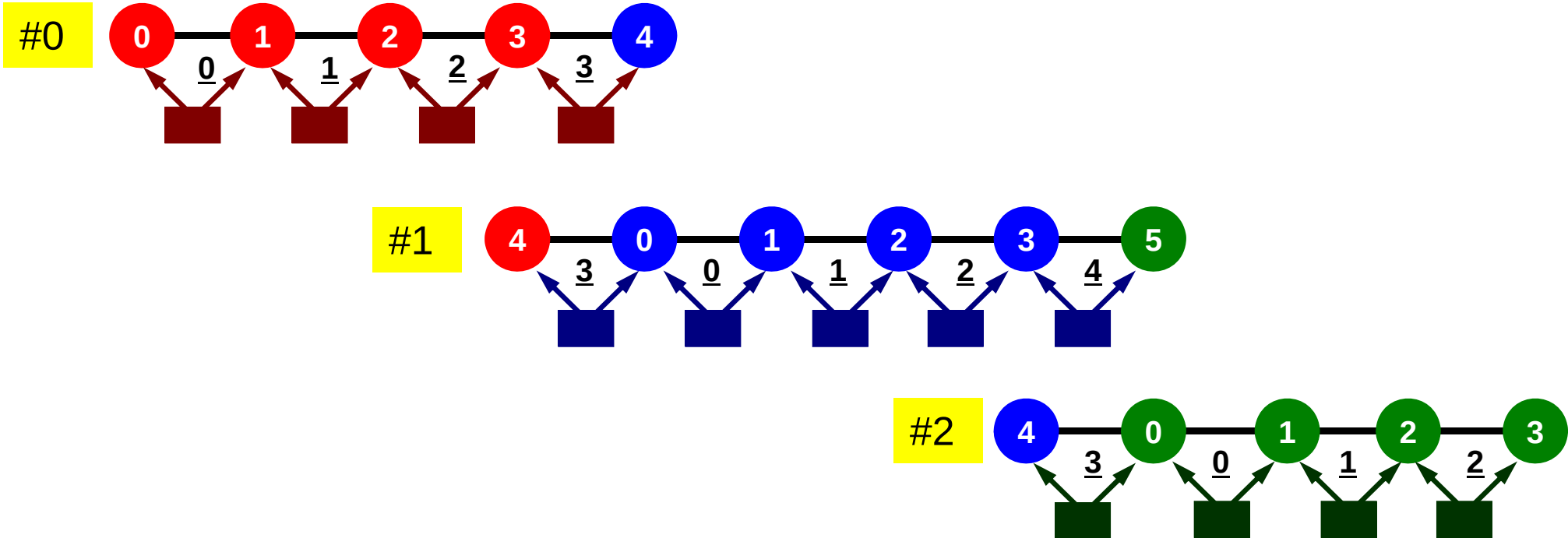
#PETot-1:
N+1 nodes
N elements



Others (General):
N+2 nodes
N+1 elements

1D FEM: 12 nodes/11 elem's/3 domains

Integration on each element, element matrix -> global matrix
Operations can be done by info. of internal/external nodes and elements which include these nodes



Preconditioned Conjugate Gradient Method (CG)

```

Compute  $\mathbf{r}^{(0)} = \mathbf{b} - [\mathbf{A}]\mathbf{x}^{(0)}$ 
for  $i = 1, 2, \dots$ 
  solve  $[\mathbf{M}]\mathbf{z}^{(i-1)} = \mathbf{r}^{(i-1)}$ 
   $\rho_{i-1} = \mathbf{r}^{(i-1)} \cdot \mathbf{z}^{(i-1)}$ 
  if  $i = 1$ 
     $\mathbf{p}^{(1)} = \mathbf{z}^{(0)}$ 
  else
     $\beta_{i-1} = \rho_{i-1} / \rho_{i-2}$ 
     $\mathbf{p}^{(i)} = \mathbf{z}^{(i-1)} + \beta_{i-1} \mathbf{p}^{(i-1)}$ 
  endif
   $\mathbf{q}^{(i)} = [\mathbf{A}]\mathbf{p}^{(i)}$ 
   $\alpha_i = \rho_{i-1} / (\mathbf{p}^{(i)} \cdot \mathbf{q}^{(i)})$ 
   $\mathbf{x}^{(i)} = \mathbf{x}^{(i-1)} + \alpha_i \mathbf{p}^{(i)}$ 
   $\mathbf{r}^{(i)} = \mathbf{r}^{(i-1)} - \alpha_i \mathbf{q}^{(i)}$ 
  check convergence  $|\mathbf{r}|$ 
end

```

Preconditioning:
Diagonal Scaling
(or Point Jacobi)

Preconditioning, DAXPY

Local Operations by Only Internal Points: Parallel Processing is possible

```
/*  
/-- {z}= [Minv]{r}  
*/  
for(i=0;i<N;i++){  
    W[Z][i] = W[DD][i] * W[R][i];  
}
```

```
/*  
/-- {x}= {x} + ALPHA*{p}  
// {r}= {r} - ALPHA*{q}  
*/  
for(i=0;i<N;i++){  
    U[i] += Alpha * W[P][i];  
    W[R][i] -= Alpha * W[Q][i];  
}
```

0
1
2
3
4
5
6
7
8
9
10
11

Dot Products

Global Summation needed: Communication ?

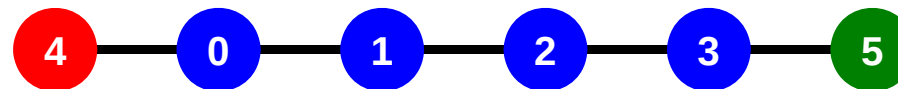
```
/*  
/-- ALPHA= RHO / {p} {q}  
*/  
C1 = 0.0;  
for (i=0; i<N; i++) {  
    C1 += W[P][i] * W[Q][i];  
}  
  
Alpha = Rho / C1;
```

0
1
2
3
4
5
6
7
8
9
10
11

Matrix-Vector Products

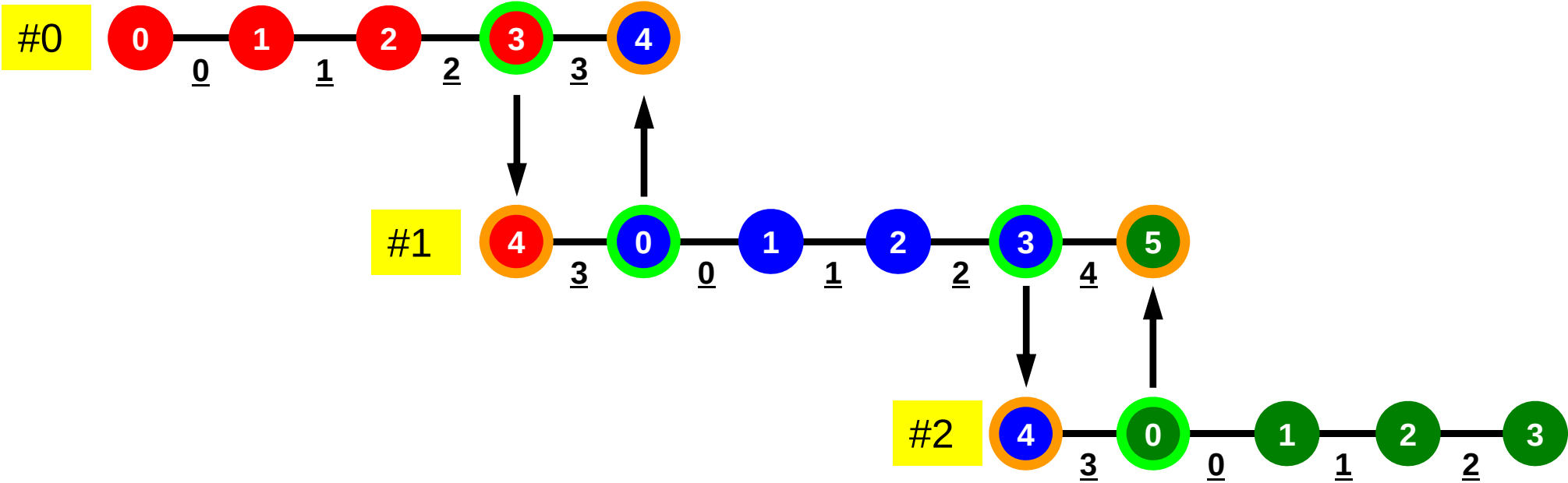
Values at External Points: P-to-P Communication

```
/*  
//-- {q} = [A] {p}  
*/  
for (i=0; i<N; i++) {  
    W[Q][i] = Diag[i] * W[P][i];  
    for (j=Index[i]; j<Index[i+1]; j++) {  
        W[Q][i] += AMat[j]*W[P][Item[j]];  
    }  
}
```

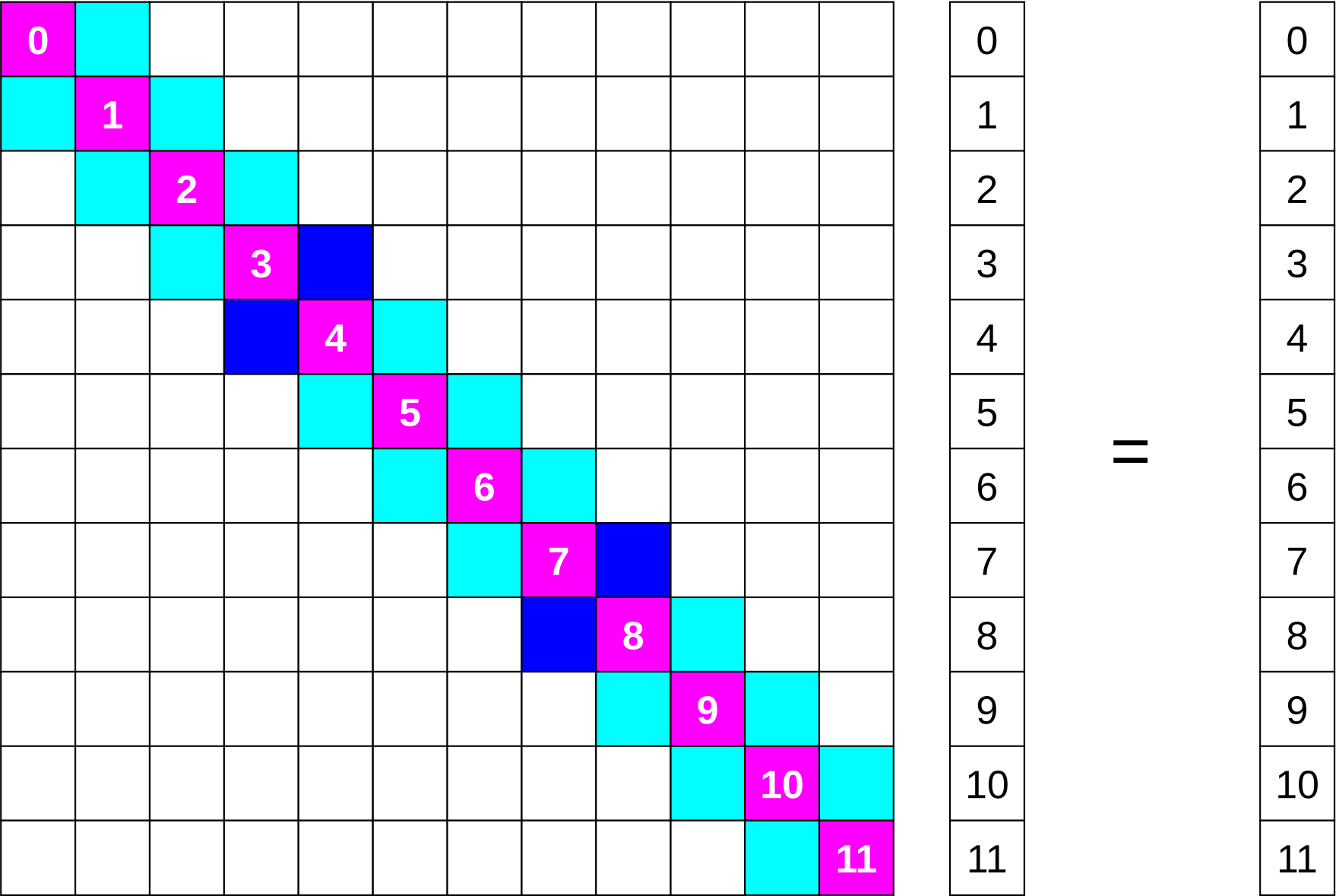


1D FEM: 12 nodes/11 elem's/3 domains

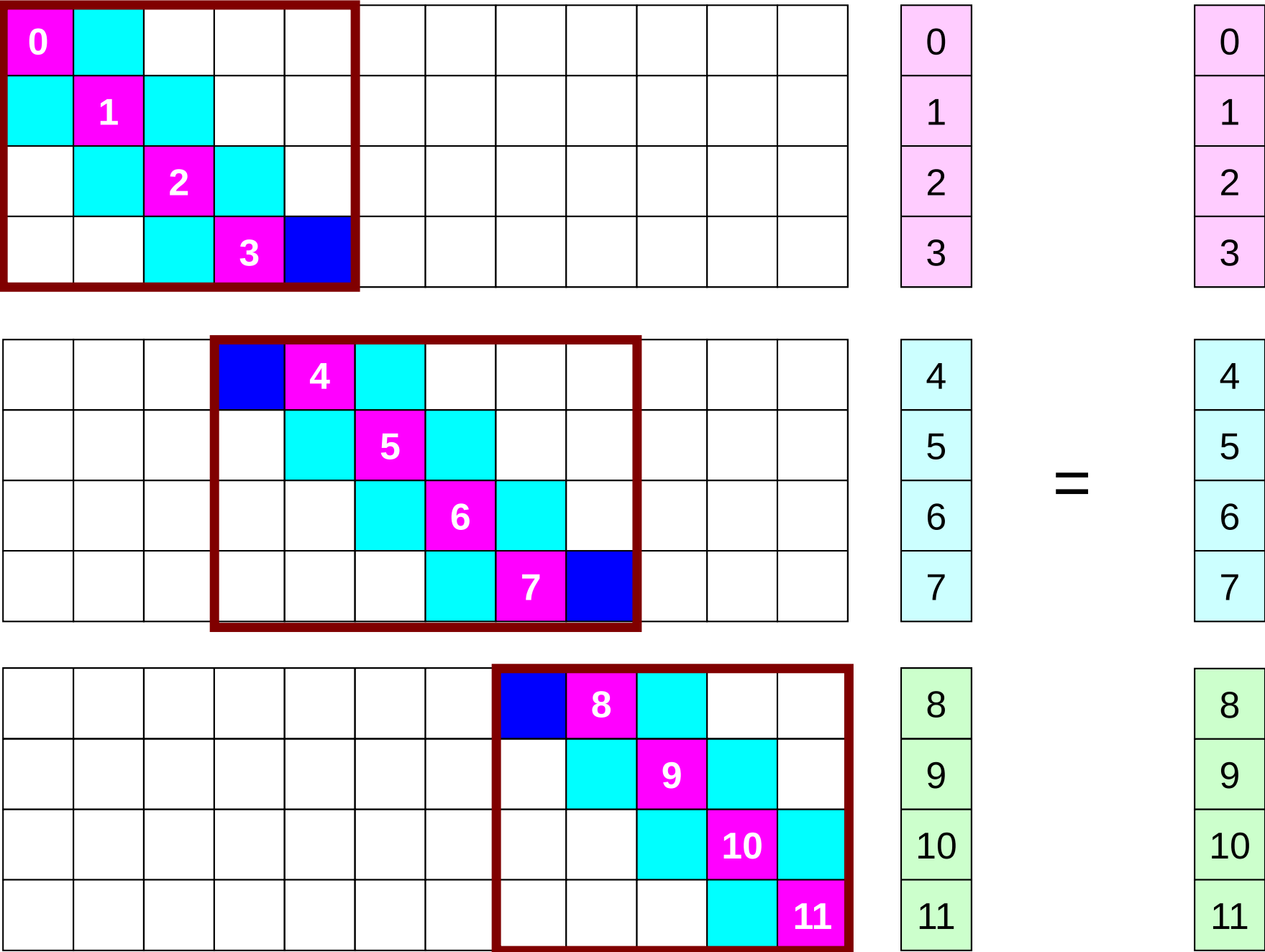
Internal/External Nodes



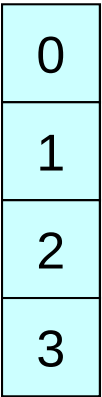
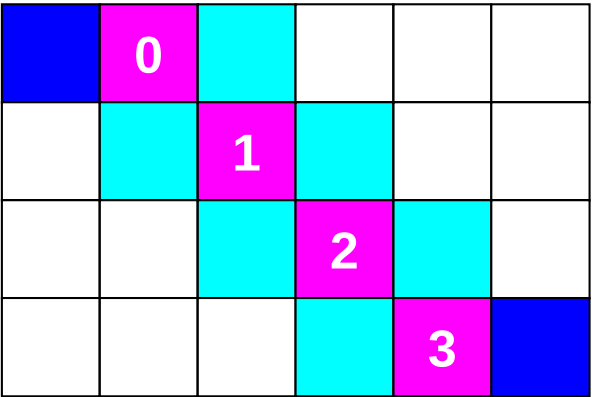
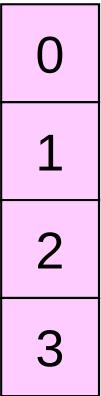
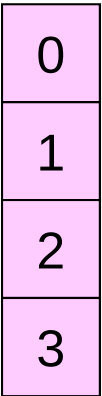
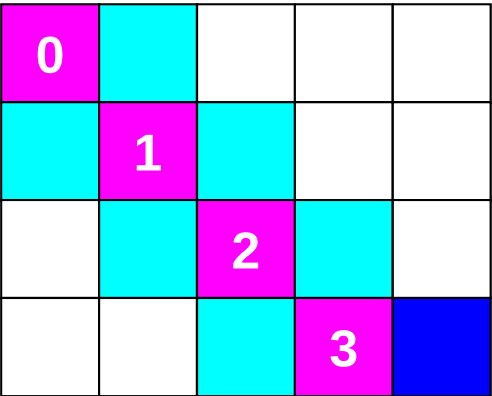
Mat-Vec Products: Local Op. Possible



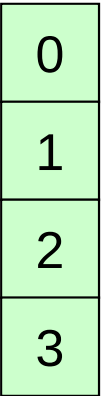
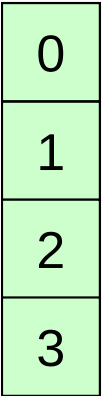
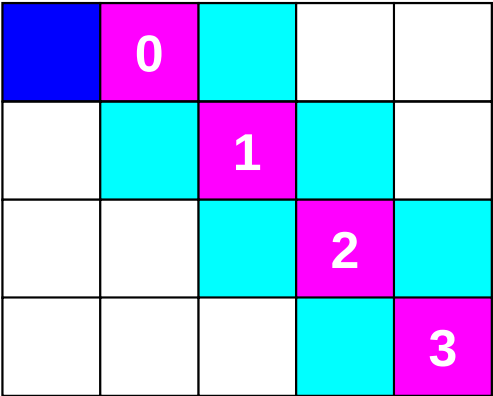
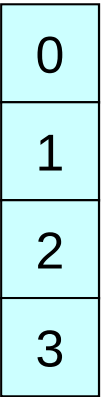
Mat-Vec Products: Local Op. Possible



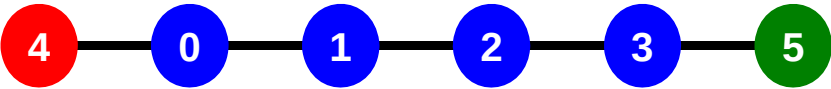
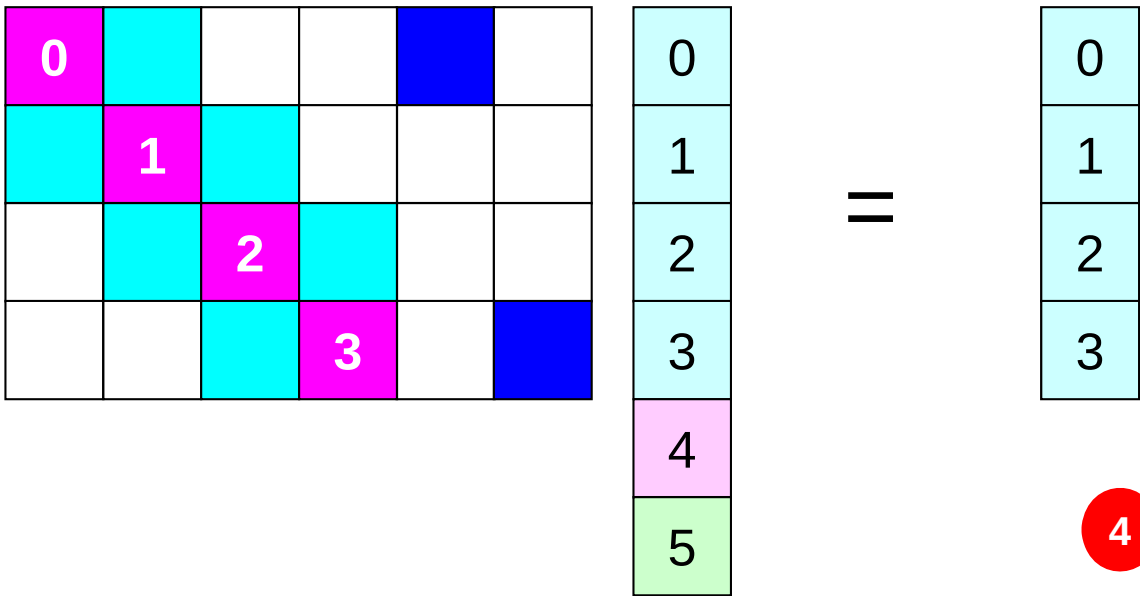
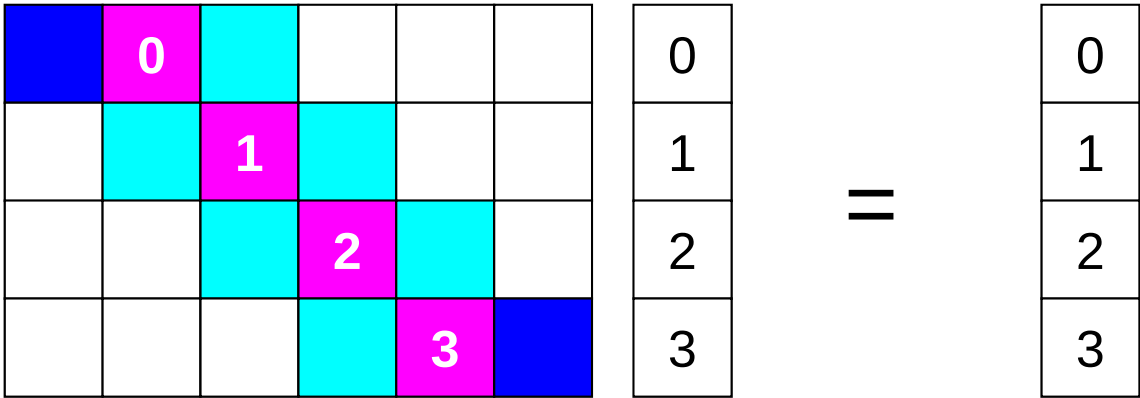
Mat-Vec Products: Local Op. Possible



=

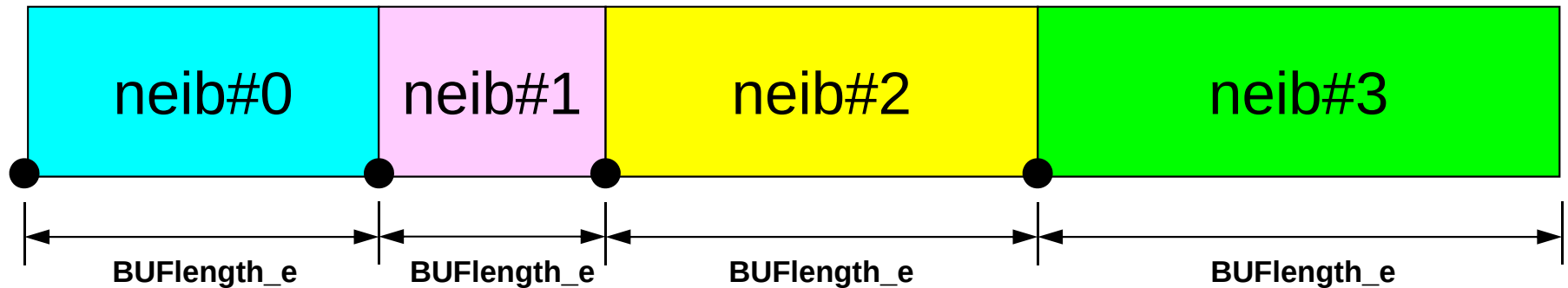


Mat-Vec Products: Local Op. #1



SEND: MPI_Isend/Irecv/Waitall

SendBuf



export_index[0] export_index[1] export_index[2] export_index[3] export_index[4]

export_item (export_index[neib]:export_index[neib+1]-1) are sent to neib-th neighbor

```
for (neib=0; neib<NeibPETot;neib++){
    for (k=export_index[neib];k<export_index[neib+1];k++){
        kk= export_item[k];
        SendBuf[k]= VAL[kk];
    }
}
```

Copied to sending buffers

```
for (neib=0; neib<NeibPETot; neib++){
    tag= 0;
    iS_e= export_index[neib];
    iE_e= export_index[neib+1];
    BUFlength_e= iE_e - iS_e

    ierr= MPI_Isend
        (&SendBuf[iS_e], BUFlength_e, MPI_DOUBLE, NeibPE[neib], 0,
         MPI_COMM_WORLD, &ReqSend[neib])
}
```

```
MPI_Waitall(NeibPETot, ReqSend, StatSend);
```

RECV: MPI_Isend/Irecv/Waitall

```
for (neib=0; neib<NeibPETot; neib++){
    tag= 0;
    iS_i= import_index[neib];
    iE_i= import_index[neib+1];
    BUFlength_i= iE_i - iS_i

    ierr= MPI_Irecv
        (&RecvBuf[iS_i], BUFlength_i, MPI_DOUBLE, NeibPE[neib], 0,
         MPI_COMM_WORLD, &ReqRecv[neib])
}

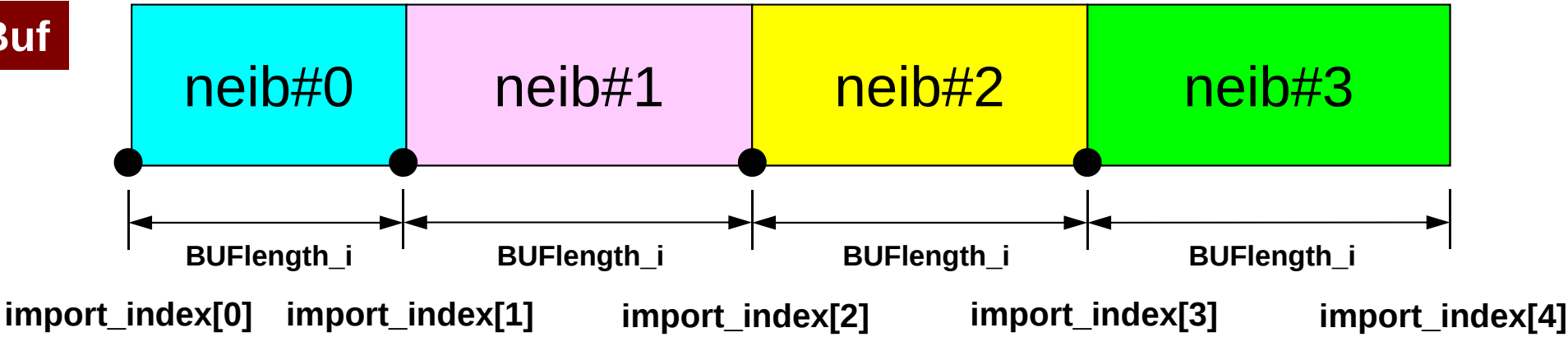
MPI_Waitall(NeibPETot, ReqRecv, StatRecv);

for (neib=0; neib<NeibPETot;neib++){
    for (k=import_index[neib];k<import_index[neib+1];k++){
        kk= import_item[k];
        VAL[kk]= RecvBuf[k];
    }
}
```

Copied from receiving buffer

import_item (import_index[neib]:import_index[neib+1]-1) are received from neib-th neighbor

RecvBuf



- Overview
- Distributed Local Data
- **Program**
- Results

Program: 1d.c (1/11)

Variables

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <assert.h>
#include <mpi.h>
```

```
int main(int argc, char **argv) {
```

Essential for programs with MPI

```
    int NE, N, NP, NPLU, IterMax, NEg, Ng, errno;
    double dX, Resid, Eps, Area, F, Young;
    double X1, X2, U1, U2, DL, Strain, Sigma, Ck;
    double *U, *Rhs, *X, *Diag, *AMat;
    double *R, *Z, *Q, *P, *DD;
    int *Index, *Item, *Icelnod;
    double Kmat[2][2], Emat[2][2];

    int i, j, in1, in2, k, icel, k1, k2, jS;
    int iter, nr, neib;
    FILE *fp;
    double BNorm2, Rho, Rho1=0.0, C1, Alpha, Beta, DNorm2;
    int PETot, MyRank, kk, is, ir, len_s, len_r, tag;
    int NeibPETot, BufLength, NeibPE[2];

    int import_index[3], import_item[2];
    int export_index[3], export_item[2];
    double SendBuf[2], RecvBuf[2];

    double BNorm20, Rho0, C10, DNorm20;
    double StartTime, EndTime;
    int ierr = 1;

    MPI_Status *StatSend, *StatRecv;
    MPI_Request *RequestSend, *RequestRecv;
```

Program: 1d.c (2/11)

Control Data

```
/*
// +-----+
// |  INIT.  |
// +-----+
//=== */
```

```
/*
//-- CONTROL data
*/
```

```
ierr = MPI_Init(&argc, &argv);
ierr = MPI_Comm_size(MPI_COMM_WORLD, &PETot);
ierr = MPI_Comm_rank(MPI_COMM_WORLD, &MyRank);
```

Initialization
Entire Process #: PETot
Rank ID (0-PETot-1): MyRank

```
if(MyRank == 0) {
    fp = fopen("input.dat", "r");
    assert(fp != NULL);
    fscanf(fp, "%d", &NEg);
    fscanf(fp, "%lf %lf %lf %lf", &dX, &F, &Area, &Young);
    fscanf(fp, "%d", &IterMax);
    fscanf(fp, "%lf", &Eps);
    fclose(fp);
}
```

```
ierr = MPI_Bcast(&NEg, 1, MPI_INT, 0, MPI_COMM_WORLD);
ierr = MPI_Bcast(&IterMax, 1, MPI_INT, 0, MPI_COMM_WORLD);
ierr = MPI_Bcast(&dX, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
ierr = MPI_Bcast(&F, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
ierr = MPI_Bcast(&Area, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
ierr = MPI_Bcast(&Young, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
ierr = MPI_Bcast(&Eps, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

Program: 1d.c (2/11)

Control Data

```
/*
// +-----+
// |  INIT.  |
// +-----+
//== */
```

```
/*
//-- CONTROL data
*/
```

```
ierr = MPI_Init(&argc, &argv);
ierr = MPI_Comm_size(MPI_COMM_WORLD, &PETot);
ierr = MPI_Comm_rank(MPI_COMM_WORLD, &MyRank);
```

Initialization
Entire Process #: PETot
Rank ID (0-PETot-1): MyRank

```
if(MyRank == 0) {
    fp = fopen("input.dat", "r");
    assert(fp != NULL);
    fscanf(fp, "%d", &NEg);
    fscanf(fp, "%lf %lf %lf %lf", &dX, &F, &Area, &Young);
    fscanf(fp, "%d", &IterMax);
    fscanf(fp, "%lf", &Eps);
    fclose(fp);
}
```

Reading control file if MyRank=0

```
ierr = MPI_Bcast(&NEg, 1, MPI_INT, 0, MPI_COMM_WORLD);
ierr = MPI_Bcast(&IterMax, 1, MPI_INT, 0, MPI_COMM_WORLD);
ierr = MPI_Bcast(&dX, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
ierr = MPI_Bcast(&F, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
ierr = MPI_Bcast(&Area, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
ierr = MPI_Bcast(&Young, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
ierr = MPI_Bcast(&Eps, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

Program: 1d.c (2/11)

Control Data

```
/*
// +-----+
// |  INIT.  |
// +-----+
//== */
```

```
/*
//-- CONTROL data
*/
```

```
ierr = MPI_Init(&argc, &argv);
ierr = MPI_Comm_size(MPI_COMM_WORLD, &PETot);
ierr = MPI_Comm_rank(MPI_COMM_WORLD, &MyRank);
```

Initialization
Entire Process #: PETot
Rank ID (0-PETot-1): MyRank

```
if(MyRank == 0) {
    fp = fopen("input.dat", "r");
    assert(fp != NULL);
    fscanf(fp, "%d", &NEg);
    fscanf(fp, "%lf %lf %lf %lf", &dX, &F, &Area, &Young);
    fscanf(fp, "%d", &IterMax);
    fscanf(fp, "%lf", &Eps);
    fclose(fp);
}
```

Reading control file if MyRank=0

Neg: Total Number of Elements (Global)

```
ierr = MPI_Bcast(&NEg, 1, MPI_INT, 0, MPI_COMM_WORLD);
ierr = MPI_Bcast(&IterMax, 1, MPI_INT, 0, MPI_COMM_WORLD);
ierr = MPI_Bcast(&dX, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
ierr = MPI_Bcast(&F, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
ierr = MPI_Bcast(&Area, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
ierr = MPI_Bcast(&Young, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
ierr = MPI_Bcast(&Eps, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

Parameters are sent to each process
from Process #0.

Diagram illustrating a Broadcast operation:

Initial State (Left):

P#0	A0	B0	C0	D0
P#1				
P#2				
P#3				

Broadcast Operation (Arrow):

Final State (Right):

P#0	A0	B0	C0	D0
P#1	A0	B0	C0	D0
P#2	A0	B0	C0	D0
P#3	A0	B0	C0	D0

- Broadcasts a message from the process with rank "root" to all other processes of the communicator
- **MPI_Bcast (buffer, count, datatype, root, comm)**
 - **buffer** choice I/O starting address of buffer
type is defined by "datatype"
 - **count** int I number of elements in send/receive buffer
 - **datatype** MPI_Datatype I data type of elements of send/recive buffer
 - FORTTRAN MPI_INTEGER, MPI_REAL, MPI_DOUBLE_PRECISION, MPI_CHARACTER etc.
 - C MPI_INT, MPI_FLOAT, MPI_DOUBLE, MPI_CHAR etc.
 - **root** int I rank of root process
 - **comm** MPI_Comm I communicator

Program: 1d.c (3/11)

Distributed Local Mesh

```

/*
/-- LOCAL MESH size
*/
Ng= NEg + 1;           Ng: Number of Nodes (Global)
N = Ng / PETot;        N : Number of Nodes (Local)

nr = Ng - N*PETot;      mod(Ng, PETot) .ne. 0
if(MyRank < nr) N++;

NE= N - 1 + 2;
NP= N + 2;
if(MyRank == 0) NE= N - 1 + 1;
if(MyRank == 0) NP= N + 1;
if(MyRank == PETot-1) NE= N - 1 + 1;
if(MyRank == PETot-1) NP= N + 1;

if(PETot==1) {NE=N-1;}
if(PETot==1) {NP=N  ;}

/*
/-- Arrays
*/
U      = calloc(NP, sizeof(double));
Diag   = calloc(NP, sizeof(double));
AMat   = calloc(2*NP-2, sizeof(double));
Rhs    = calloc(NP, sizeof(double));
Index  = calloc(NP+1, sizeof(int));
Item   = calloc(2*NP-2, sizeof(int));
Icelnod= calloc(2*NE, sizeof(int));

```

Program: 1d.c (3/11)

Distributed Local Mesh, Uniform Elements

```

/*
  -- LOCAL MESH size
*/
Ng= NEg + 1;           Ng: Number of Nodes (Global)
N = Ng / PETot;        N : Number of Nodes (Local)

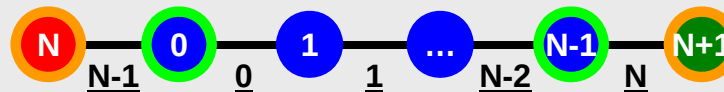
nr = Ng - N*PETot;     mod(Ng, PETot) .ne. 0
if(MyRank < nr) N++;

NE= N - 1 + 2;         Number of Elements (Local)
NP= N + 2;             Total Number of Nodes (Local) (Internal + External Nodes)
if(MyRank == 0) NE= N - 1 + 1;
if(MyRank == 0) NP= N + 1;
if(MyRank == PETot-1) NE= N - 1 + 1;
if(MyRank == PETot-1) NP= N + 1;

if(PETot==1) {NE=N-1;}
if(PETot==1) {NP=N ;}

/*
  -- Arrays
*/
U      = calloc(NP, sizeof(double));
Diag   = calloc(NP, sizeof(double));
AMat   = calloc(2*NP-2, sizeof(double));
Rhs    = calloc(NP, sizeof(double));
Index  = calloc(NP+1, sizeof(int));
Item   = calloc(2*NP-2, sizeof(int));
Icelnod= calloc(2*NE, sizeof(int));

```



Others (General):
 N+2 nodes
 N+1 elements

Program: 1d.c (3/11)

Distributed Local Mesh, Uniform Elements

```

/*
/-- LOCAL MESH size
*/
Ng= NEg + 1;           Ng: Number of Nodes (Global)
N = Ng / PETot;        N : Number of Nodes (Local)

```

```

nr = Ng - N*PETot;      mod(Ng, PETot) .ne. 0
if(MyRank < nr) N++;

```

```

NE= N - 1 + 2;
NP= N + 2;
if(MyRank == 0) NE= N - 1 + 1;
if(MyRank == 0) NP= N + 1;
if(MyRank == PETot-1) NE= N - 1 + 1;
if(MyRank == PETot-1) NP= N + 1;

```

```

if(PETot==1) {NE=N-1;}
if(PETot==1) {NP=N ;}

```

```

/*
/-- Arrays
*/

```

```

U      = calloc(NP, sizeof(double));
Diag   = calloc(NP, sizeof(double));
AMat   = calloc(2*NP-2, sizeof(double));
Rhs    = calloc(NP, sizeof(double));
Index  = calloc(NP+1, sizeof(int));
Item   = calloc(2*NP-2, sizeof(int));
Icelnod= calloc(2*NE, sizeof(int));

```



#0:
N+1 nodes
N elements

Program: 1d.c (3/11)

Distributed Local Mesh, Uniform Elements

```

/*
  -- LOCAL MESH size
*/
Ng= NEg + 1;           Ng: Number of Nodes (Global)
N = Ng / PETot;        N : Number of Nodes (Local)

nr = Ng - N*PETot;      mod(Ng, PETot) .ne. 0
if(MyRank < nr) N++;
NE= N - 1 + 2;
NP= N + 2;
if(MyRank == 0) NE= N - 1 + 1;
if(MyRank == 0) NP= N + 1;
if(MyRank == PETot-1) NE= N - 1 + 1;
if(MyRank == PETot-1) NP= N + 1;

if(PETot==1) {NE=N-1;}
if(PETot==1) {NP=N ;}

/*
  -- Arrays
*/
U      = calloc(NP, sizeof(double));
Diag   = calloc(NP, sizeof(double));
AMat   = calloc(2*NP-2, sizeof(double));
Rhs    = calloc(NP, sizeof(double));
Index  = calloc(NP+1, sizeof(int));
Item   = calloc(2*NP-2, sizeof(int));
Icelnod= calloc(2*NE, sizeof(int));

```



#PETot-1:
N+1 nodes
N elements

Program: 1d.c (3/11)

Distributed Local Mesh, Uniform Elements

```

/*
/-- LOCAL MESH size
*/
Ng= NEg + 1;           Ng: Number of Nodes (Global)
N = Ng / PETot;        N : Number of Nodes (Local)

nr = Ng - N*PETot;      mod(Ng, PETot) .ne. 0
if(MyRank < nr) N++;

NE= N - 1 + 2;
NP= N + 2;
if(MyRank == 0) NE= N - 1 + 1;
if(MyRank == 0) NP= N + 1;
if(MyRank == PETot-1) NE= N - 1 + 1;
if(MyRank == PETot-1) NP= N + 1;

if(PETot==1) {NE=N-1;}
if(PETot==1) {NP=N  ;}

```

```

/*
/-- Arrays
*/

```

```

U      = calloc(NP, sizeof(double));
Diag   = calloc(NP, sizeof(double));
AMat   = calloc(2*NP-2, sizeof(double));
Rhs    = calloc(NP, sizeof(double));
Index  = calloc(NP+1, sizeof(int));
Item   = calloc(2*NP-2, sizeof(int));
Icelnod= calloc(2*NE, sizeof(int));

```

Size of arrays is "NP" , not "N"

Program: 1d.c (4/11)

Initialization of Arrays, Elements-Nodes

```

for (i=0; i<NP; i++)    U[i] = 0.0;
for (i=0; i<NP; i++)    Diag[i] = 0.0;
for (i=0; i<NP; i++)    Rhs[i] = 0.0;
for (k=0; k<2*NP-2; k++) AMat[k] = 0.0;

```

```

for (i=0; i<3; i++) import_index[i]= 0;
for (i=0; i<3; i++) export_index[i]= 0;
for (i=0; i<2; i++) import_item[i]= 0;
for (i=0; i<2; i++) export_item[i]= 0;

```

```

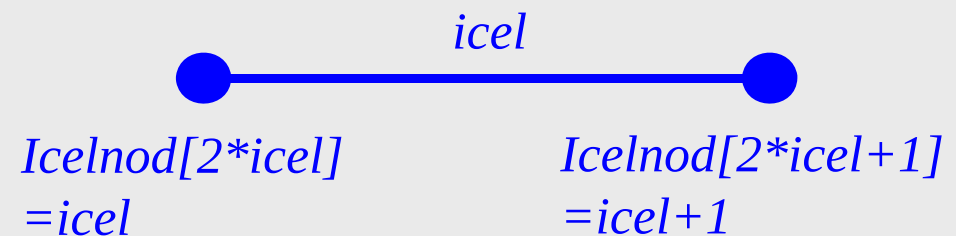
for (icel=0; icel<NE; icel++) {
    Icelnod[2*icel] = icel;
    Icelnod[2*icel+1] = icel+1;
}

```

```

if (PETot>1) {
    if (MyRank==0) {
        icel = NE-1;
        Icelnod[2*icel] = N-1;
        Icelnod[2*icel+1] = N;
    } else if (MyRank==PETot-1) {
        icel = NE-1;
        Icelnod[2*icel] = N;
        Icelnod[2*icel+1] = 0;
    } else {
        icel = NE-2;
        Icelnod[2*icel] = N;
        Icelnod[2*icel+1] = 0;
        icel = NE-1;
        Icelnod[2*icel] = N-1;
        Icelnod[2*icel+1] = N+1;
    }
}
}

```



Program: 1d.c (4/11)

Initialization of Arrays, Elements-Nodes

```

for (i=0; i<NP; i++)    U[i] = 0.0;
for (i=0; i<NP; i++)    Diag[i] = 0.0;
for (i=0; i<NP; i++)    Rhs[i] = 0.0;
for (k=0; k<2*NP-2; k++) AMat[k] = 0.0;

```

```

for (i=0; i<3; i++) import_index[i]= 0;
for (i=0; i<3; i++) export_index[i]= 0;
for (i=0; i<2; i++) import_item[i]= 0;
for (i=0; i<2; i++) export_item[i]= 0;

```

```

for (icel=0; icel<NE; icel++) {
    Icelnod[2*icel] = icel;
    Icelnod[2*icel+1] = icel+1;
}

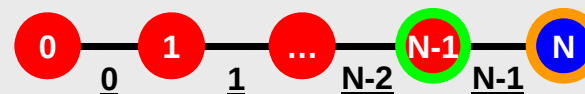
```

```

if (PETot>1) {
    if (MyRank==0) {
        icel = NE-1;
        Icelnod[2*icel] = N-1;
        Icelnod[2*icel+1] = N;
    } else if (MyRank==PETot-1) {
        icel = NE-1;
        Icelnod[2*icel] = N;
        Icelnod[2*icel+1] = 0;
    } else {
        icel = NE-2;
        Icelnod[2*icel] = N;
        Icelnod[2*icel+1] = 0;
        icel = NE-1;
        Icelnod[2*icel] = N-1;
        Icelnod[2*icel+1] = N+1;
    }
}

```

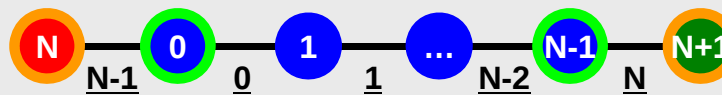
e.g. Element-0 includes node-0 and node-1



#0:
N+1 nodes
N elements



#PETot-1:
N+1 nodes
N elements



Others (General):
N+2 nodes
N+1 elements

Program: 1d.c (5/11)

"Index"

```
Kmat[0][0]= +1.0;  
Kmat[0][1]= -1.0;  
Kmat[1][0]= -1.0;  
Kmat[1][1]= +1.0;  
  
/*  
// +-----+  
// | CONNECTIVITY |  
// +-----+  
*/  
  
for (i=0; i<N+1; i++) Index[i] = 2;  
for (i=N+1; i<NP+1; i++) Index[i] = 1;  
  
Index[0] = 0;  
if (MyRank == 0) Index[1] = 1;  
if (MyRank == PETot-1) Index[N] = 1;  
  
for (i=0; i<NP; i++) {  
    Index[i+1]= Index[i+1] + Index[i];  
}  
  
NPLU= Index[NP];
```

#0:
N+1 nodes
N elements

#PETot-1:
N+1 nodes
N elements

Others (General):
N+2 nodes
N+1 elements

Program: 1d.c (6/11)

"Item"

```
for (i=0; i<N; i++) {  
    jS = Index[i];  
    if ((MyRank==0)&&(i==0)) {  
        Item[jS] = i+1;  
    } else if ((MyRank==PETot-1)&&(i==N-1)) {  
        Item[jS] = i-1;  
    } else {  
        Item[jS] = i-1;  
        Item[jS+1] = i+1;  
        if (i==0) { Item[jS] = N;}  
        if (i==N-1) { Item[jS+1] = N+1;}  
        if ((MyRank==0)&&(i==N-1)) {Item[jS+1] = N;}  
    }  
}
```

```
i =N;  
jS= Index[i];  
if (MyRank==0) {  
    Item[jS]= N-1;  
} else {  
    Item[jS]= 0;  
}
```

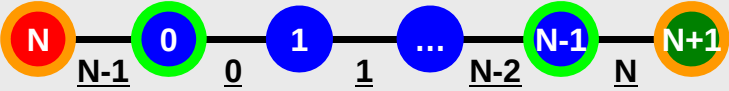
```
i =N+1;  
jS= Index[i];  
if ((MyRank!=0)&&(MyRank!=PETot-1)) {  
    Item[jS]= N-1;  
}
```



#0:
N+1 nodes
N elements



#PETot-1:
N+1 nodes
N elements

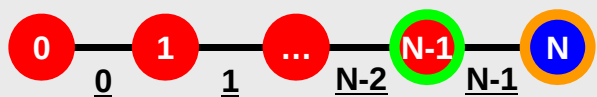


Others (General):
N+2 nodes
N+1 elements

Program: 1d.c (7/11)

Communication Tables

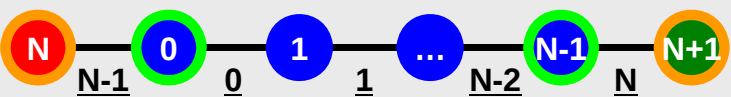
```
/*  
//-- COMMUNICATION  
*/  
NeibPETot = 2;  
if(MyRank == 0) NeibPETot = 1;  
if(MyRank == PETot-1) NeibPETot = 1;  
if(PETot == 1) NeibPETot = 0;  
  
NeibPE[0] = MyRank - 1;  
NeibPE[1] = MyRank + 1;  
  
if(MyRank == 0) NeibPE[0] = MyRank + 1;  
if(MyRank == PETot-1) NeibPE[0] = MyRank - 1;  
  
import_index[1]=1;  
import_index[2]=2;  
import_item[0]= N;  
import_item[1]= N+1;  
  
export_index[1]=1;  
export_index[2]=2;  
export_item[0]= 0;  
export_item[1]= N-1;  
  
if(MyRank == 0) import_item[0]=N;  
if(MyRank == 0) export_item[0]=N-1;  
  
BufLength = 1;  
  
StatSend = malloc(sizeof(MPI_Status) * NeibPETot);  
StatRecv = malloc(sizeof(MPI_Status) * NeibPETot);  
RequestSend = malloc(sizeof(MPI_Request) * NeibPETot);  
RequestRecv = malloc(sizeof(MPI_Request) * NeibPETot);
```



#0:
N+1 nodes
N elements



#PETot-1:
N+1 nodes
N elements



Others (General):
N+2 nodes
N+1 elements

MPI_Isend

- Begins a non-blocking send
 - Send the contents of sending buffer (starting from **sendbuf**, number of messages: **count**) to **dest** with **tag** .
 - Contents of sending buffer cannot be modified before calling corresponding **MPI_Waitall**.

- **MPI_Isend**

(sendbuf, count, datatype, dest, tag, comm, request)

- sendbuf choice I starting address of sending buffer
- count int I number of elements in sending buffer
- datatype MPI_Datatype I datatype of each sending buffer element
- dest int I rank of destination
- tag int I message tag

This integer can be used by the application to distinguish messages. Communication occurs if tag's of MPI_Isend and MPI_Irecv are matched. Usually tag is set to be "0" (in this class),
- comm MPI_Comm I communicator
- request MPI_Request 0 communication request array used in MPI_Waitall

MPI_Irecv

- Begins a non-blocking receive
 - Receiving the contents of receiving buffer (starting from **recvbuf**, number of messages: **count**) from **source** with **tag** .
 - Contents of receiving buffer cannot be used before calling corresponding **MPI_Waitall**.

- **MPI_Irecv**
(recvbuf, count, datatype, source, tag, comm, request)

- **recvbuf** choice I starting address of receiving buffer
- **count** int I number of elements in receiving buffer
- **datatype** MPI_Datatype I datatype of each receiving buffer element
- **source** int I rank of source
- **tag** int I message tag

This integer can be used by the application to distinguish messages. Communication occurs if tag's of MPI_Isend and MPI_Irecv are matched. Usually tag is set to be "0" (in this class),
- **comm** MPI_Comm I communicator
- **request** MPI_Request 0 communication request array used in MPI_Waitall

MPI_Waitall

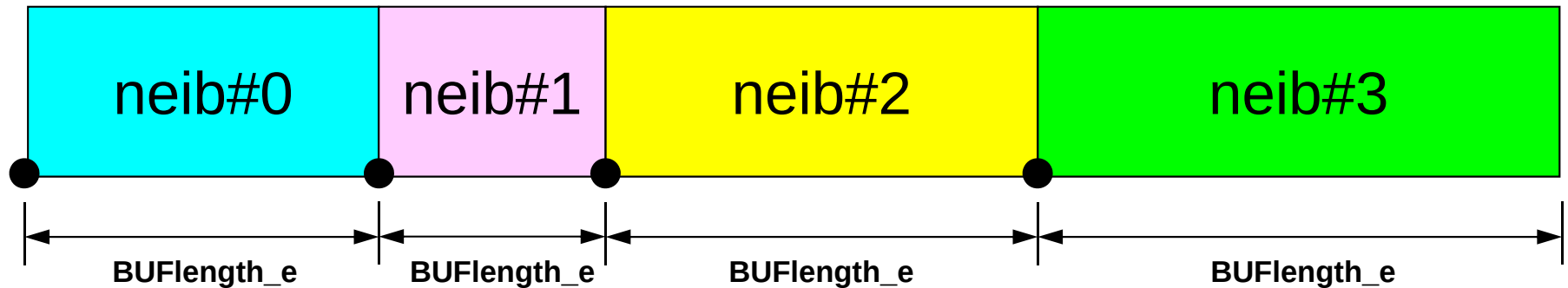
- **MPI_Waitall** blocks until all comm's, associated with request in the array, complete. It is used for synchronizing **MPI_Isend** and **MPI_Irecv** in this class.
- At sending phase, contents of sending buffer cannot be modified before calling corresponding **MPI_Waitall**. At receiving phase, contents of receiving buffer cannot be used before calling corresponding **MPI_Waitall**.
- **MPI_Isend** and **MPI_Irecv** can be synchronized simultaneously with a single **MPI_Waitall** if it is consistent.
 - Same request should be used in **MPI_Isend** and **MPI_Irecv**.
- Its operation is similar to that of **MPI_Barrier** but, **MPI_Waitall** can not be replaced by **MPI_Barrier**.
 - Possible troubles using **MPI_Barrier** instead of **MPI_Waitall**: Contents of **request** and **status** are not updated properly, very slow operations etc.
- **MPI_Waitall (count, request, status)**
 - count int I number of processes to be synchronized
 - request MPI_Request I/O comm. request used in MPI_Waitall (array size: count)
 - status MPI_Status 0 array of status objects
MPI_STATUS_SIZE: defined in 'mpif.h', 'mpi.h'

Generalized Comm. Table: Send

- Neighbors
 - NeibPETot, NeibPE[neib]
- Message size for each neighbor
 - export_index[neib], neib= 0, NeibPETot
- ID of **boundary** points
 - export_item[k], k= 0, export_index[NeibPETot]-1
- Messages to each neighbor
 - SendBuf[k], k= 0, export_index[NeibPETot]-1

SEND: MPI_Isend/Irecv/Waitall

SendBuf



export_index[0] export_index[1] export_index[2] export_index[3] export_index[4]

export_item (export_index[neib]:export_index[neib+1]-1) are sent to neib-th neighbor

```
for (neib=0; neib<NeibPETot;neib++){
    for (k=export_index[neib];k<export_index[neib+1];k++){
        kk= export_item[k];
        SendBuf[k]= VAL[kk];
    }
}
```

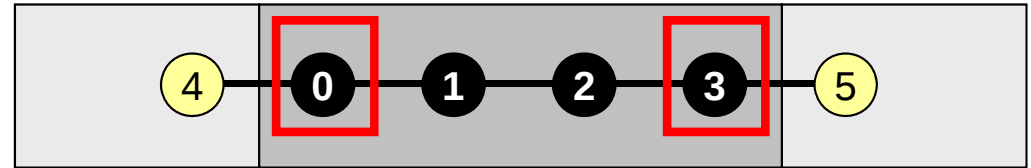
Copied to sending buffers

```
for (neib=0; neib<NeibPETot; neib++){
    tag= 0;
    iS_e= export_index[neib];
    iE_e= export_index[neib+1];
    BUFlength_e= iE_e - iS_e

    ierr= MPI_Isend
        (&SendBuf[iS_e], BUFlength_e, MPI_DOUBLE, NeibPE[neib], 0,
         MPI_COMM_WORLD, &ReqSend[neib])
}
```

```
MPI_Waitall(NeibPETot, ReqSend, StatSend);
```

SEND/Export: 1D Problem



SendBuf[0]=VAL[0]

SendBuf[1]=VAL[3]

- Neighbors
 - NeibPETot, NeibPE[neib]
 - NeibPETot=2, NeibPE[0]= my_rank-1, NeibPE[1]= my_rank+1
- Message size for each neighbor
 - export_index[neib], neib= 0, NeibPETot
 - export_index[0]=0, export_index[1]= 1, export_index[2]= 2
- ID of **boundary** points
 - export_item[k], k= 0, export_index[NeibPETot]-1
 - export_item[0]= 0, export_item[1]= N-1
- Messages to each neighbor
 - SendBuf[k], k= 0, export_index[NeibPETot]-1
 - SendBuf[0]= VAL[0], SendBuf[1]= VAL[N-1]

Generalized Comm. Table: Receive

- Neighbors
 - NeibPETot , NeibPE[neib]
- Message size for each neighbor
 - import_index[neib], neib= 0, NeibPETot
- ID of **external** points
 - import_item[k], k= 0, import_index[NeibPETot]-1
- Messages from each neighbor
 - RecvBuf[k], k= 0, import_index[NeibPETot]-1

RECV: MPI_Isend/Irecv/Waitall

```
for (neib=0; neib<NeibPETot; neib++){
    tag= 0;
    iS_i= import_index[neib];
    iE_i= import_index[neib+1];
    BUFlength_i= iE_i - iS_i

    ierr= MPI_Irecv
        (&RecvBuf[iS_i], BUFlength_i, MPI_DOUBLE, NeibPE[neib], 0,
         MPI_COMM_WORLD, &ReqRecv[neib])
}

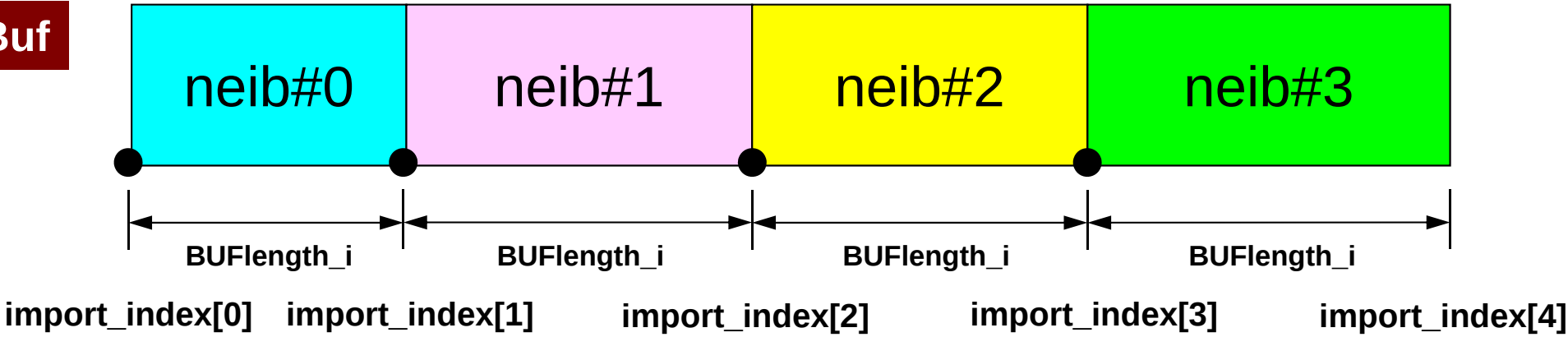
MPI_Waitall(NeibPETot, ReqRecv, StatRecv);

for (neib=0; neib<NeibPETot;neib++){
    for (k=import_index[neib];k<import_index[neib+1];k++){
        kk= import_item[k];
        VAL[kk]= RecvBuf[k];
    }
}
```

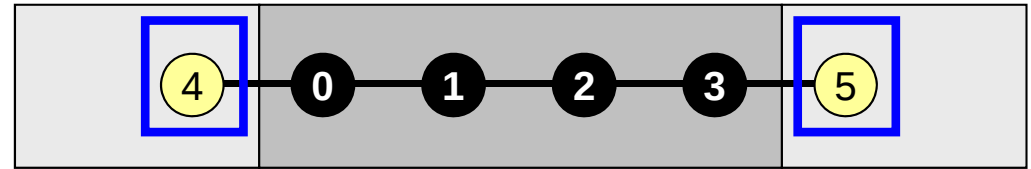
Copied from receiving buffer

import_item (import_index[neib]:import_index[neib+1]-1) are received from neib-th neighbor

RecvBuf



RECV/Import: 1D Problem



- Neighbors
 - NeibPETot, NeibPE[neib]
 - NeibPETot=2, NeibPE[0]= my_rank-1, NeibPE[1]= my_rank+1
- Message size for each neighbor
 - import_index[neib], neib= 0, NeibPETot
 - import_index[0]=0, import_index[1]= 1, import_index[2]= 2
- ID of external points
 - import_item[k], k= 0, import_index[NeibPETot]-1
 - import_item[0]= N, import_item[1]= N+1
- Messages from each neighbor
 - RECVbuf[k], k= 0, import_index[NeibPETot]-1
 - VAL[N]=RecvBuf[0], VAL[N+1]=RecvBuf[1]

Program: 1d.c (8/11)

Matrix Assembling, NO changes from 1-CPU code

```
/*
// -----
// | MATRIX assemble |
// -----
*/
for (icel=0; icel<NE; icel++) {
    in1= Icelnod[2*icel];
    in2= Icelnod[2*icel+1];
    DL = dX;

    Ck= Area*Young/DL;
    Emat[0][0]= Ck*Kmat[0][0];
    Emat[0][1]= Ck*Kmat[0][1];
    Emat[1][0]= Ck*Kmat[1][0];
    Emat[1][1]= Ck*Kmat[1][1];

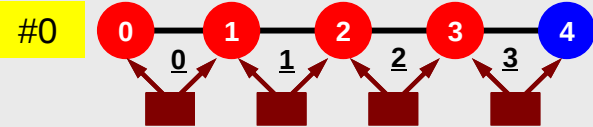
    Diag[in1]= Diag[in1] + Emat[0][0];
    Diag[in2]= Diag[in2] + Emat[1][1];

    if ((MyRank==0)&&(icel==0)) {
        k1=Index[in1];
    }else {k1=Index[in1]+1;}

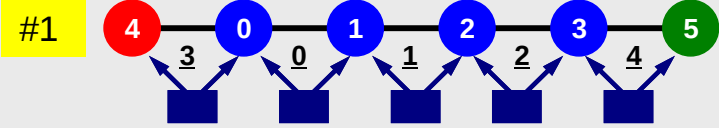
    k2=Index[in2];

    AMat[k1]= AMat[k1] + Emat[0][1];
    AMat[k2]= AMat[k2] + Emat[1][0];
}
```

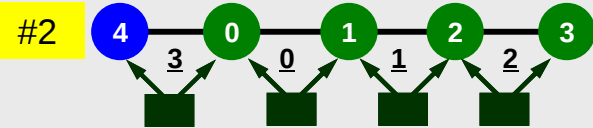
#0



#1



#2



Program: 1d.c (9/11)

Boundary Cond., ALMOST NO changes from 1-CPU code

```
/*
// |-----|
// | BOUNDARY conditions |
// |-----|
*/

/* X=Xmin */
if (MyRank==0) {
    i=0;
    jS= Index[i];
    AMat[jS]= 0.0;
    Diag[i]= 1.0;
    Rhs [i]= 0.0;

    for (k=0;k<NPLU;k++) {
        if (Item[k]==0) {AMat[k]=0.0;}
    }
}

/* X=Xmax */
if (MyRank==PETot-1) {
    i=N-1;
    Rhs[i]= F;
}
```

The diagram illustrates the domain decomposition for a 1D problem across three processors, labeled #0, #1, and #2. Each processor manages a sequence of nodes (represented by circles) and their associated indices (represented by underlined numbers below the nodes).

- Processor #0:** Contains nodes 0, 1, 2, 3, and 4. The indices below the nodes are 0, 1, 2, 3, and 4 respectively. Nodes 3 and 4 are highlighted with a green circle and an orange border. An arrow points from node 3 to node 4 in processor #1.
- Processor #1:** Contains nodes 4, 0, 1, 2, 3, and 5. The indices below the nodes are 3, 0, 1, 2, 4, and 5 respectively. Nodes 4 and 0 are highlighted with a green circle and an orange border. An arrow points from node 0 to node 3 in processor #2.
- Processor #2:** Contains nodes 4, 0, 1, 2, and 3. The indices below the nodes are 3, 0, 1, 2, and 3 respectively. Nodes 4 and 0 are highlighted with a green circle and an orange border.

The diagram shows the flow of data between adjacent processors at the interfaces. Processor #0 sends data to processor #1 at the interface between nodes 3 and 4. Processor #1 sends data to processor #2 at the interface between nodes 0 and 3.

Program: 1d.c(10/11)

Conjugate Gradient Method

```

/*
// +-----+
// | CG iterations |
// +-----+
//=== */
R = calloc(NP, sizeof(double));
Z = calloc(NP, sizeof(double));
P = calloc(NP, sizeof(double));
Q = calloc(NP, sizeof(double));
DD= calloc(NP, sizeof(double));

for(i=0;i<N;i++){
    DD[i]= 1.0 / Diag[i];
}

/*
//-- {r0}= {b} - [A]{xini} |
*/
for(neib=0;neib<NeibPETot;neib++){
    for(k=export_index[neib];k<export_index[neib+1];k++){
        kk= export_item[k];
        SendBuf[k]= U[kk];
    }
}

```

```

Compute  $r^{(0)} = b - [A]x^{(0)}$ 
for i= 1, 2, ...
    solve  $[M]z^{(i-1)} = r^{(i-1)}$ 
     $\rho_{i-1} = r^{(i-1)} z^{(i-1)}$ 
    if i=1
         $p^{(1)} = z^{(0)}$ 
    else
         $\beta_{i-1} = \rho_{i-1} / \rho_{i-2}$ 
         $p^{(i)} = z^{(i-1)} + \beta_{i-1} p^{(i-1)}$ 
    endif
     $q^{(i)} = [A]p^{(i)}$ 
     $\alpha_i = \rho_{i-1} / p^{(i)} q^{(i)}$ 
     $x^{(i)} = x^{(i-1)} + \alpha_i p^{(i)}$ 
     $r^{(i)} = r^{(i-1)} - \alpha_i q^{(i)}$ 
    check convergence |r|
end

```

Conjugate Gradient Method (CG)

- Matrix-Vector Multiply
- Dot Product
- Preconditioning: in the same way as 1CPU code
- DAXPY: in the same way as 1CPU code

Preconditioning, DAXPY

```
/*  
//-- {z} = [Minv] {r}  
*/  
    for (i=0; i<N; i++) {  
        Z[i] = DD[i] * R[i];  
    }
```

```
/*  
//-- {x} = {x} + ALPHA*{p}  
//   {r} = {r} - ALPHA*{q}  
*/  
    for (i=0; i<N; i++) {  
        U[i] += Alpha * P[i];  
        R[i] -= Alpha * Q[i];  
    }
```

Matrix-Vector Multiply (1/2)

Using Comm. Table, {p} is updated before computation

```
/*  
/-- {q} = [A] {p}  
*/  
for(neib=0;neib<NeibPETot;neib++) {  
    for(k=export_index[neib];k<export_index[neib+1];k++) {  
        kk= export_item[k];  
        SendBuf[k]= P[kk];  
    }  
}  
  
for(neib=0;neib<NeibPETot;neib++) {  
    is = export_index[neib];  
    len_s= export_index[neib+1] - export_index[neib];  
    MPI_Isend(&SendBuf[is], len_s, MPI_DDOUBLE, NeibPE[neib],  
             0, MPI_COMM_WORLD, &RequestSend[neib]);  
}  
  
for(neib=0;neib<NeibPETot;neib++) {  
    ir = import_index[neib];  
    len_r= import_index[neib+1] - import_index[neib];  
    MPI_Irecv(&RecvBuf[ir], len_r, MPI_DDOUBLE, NeibPE[neib],  
             0, MPI_COMM_WORLD, &RequestRecv[neib]);  
}  
  
MPI_Waitall(NeibPETot, RequestRecv, StatRecv);  
  
for(neib=0;neib<NeibPETot;neib++) {  
    for(k=import_index[neib];k<import_index[neib+1];k++) {  
        kk= import_item[k];  
        P[kk]=RecvBuf[k];  
    }  
}
```

Matrix-Vector Multiply (2/2)

$$\{q\} = [A]\{p\}$$

```
MPI_Waitall(NeibPETot, RequestSend, StatSend);
```

```
for (i=0; i<N; i++) {  
    Q[i] = Diag[i] * P[i];  
    for (j=Index[i]; j<Index[i+1]; j++) {  
        Q[i] += AMat[j]*P[Item[j]];  
    }  
}
```

Dot Product

Global Summation by MPI_Allreduce

```
/*  
//-- RH0= {r} {z}  
*/  
    Rho0= 0.0;  
    for (i=0; i<N; i++) {  
        Rho0 += R[i] * Z[i];  
    }  
  
    ierr = MPI_Allreduce(&Rho0, &Rho, 1, MPI_DOUBLE,  
                        MPI_SUM, MPI_COMM_WORLD);
```

The diagram illustrates the Reduce operation. On the left, a 4x4 grid of data is shown with rows P#0 to P#3 and columns A0 to D0. A large gray arrow labeled "Reduce" points to the right. On the right, a 4x4 grid of output is shown with rows P#0 to P#3 and columns op.A0-A3 to op.D0-D3. The output grid is currently empty.

P#0	A0	B0	C0	D0
P#1	A1	B1	C1	D1
P#2	A2	B2	C2	D2
P#3	A3	B3	C3	D3

Reduce

P#0	op.A0-A3	op.B0-B3	op.C0-C3	op.D0-D3
P#1				
P#2				
P#3				

- Reduces values on all processes to a single value
 - Summation, Product, Max, Min etc.
- **MPI_Reduce (sendbuf, recvbuf, count, datatype, op, root, comm)**
 - **sendbuf** choice I starting address of send buffer
 - **recvbuf** choice 0 starting address receive buffer
 - **count** int I number of elements in send/receive buffer
 - **datatype** MPI_Datatype I data type of elements of send/recive buffer
 - FORTTRAN MPI_INTEGER, MPI_REAL, MPI_DOUBLE_PRECISION, MPI_CHARACTER etc.
 - C MPI_INT, MPI_FLOAT, MPI_DOUBLE, MPI_CHAR etc
 - **op** MPI_Op I reduce operation
 - MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD, MPI_LAND, MPI_BAND etc

Users can define operations by **MPI_OP_CREATE**
 - **root** int I rank of root process
 - **comm** MPI_Comm I communicator

Send/Receive Buffer (Sending/Receiving)

- Arrays of “send (sending) buffer” and “receive (receiving) buffer” often appear in MPI.
- Addresses of “send (sending) buffer” and “receive (receiving) buffer” must be different.

Example of MPI_Reduce (1/2)

```
call MPI_REDUCE  
(sendbuf, recvbuf, count, datatype, op, root, comm, ierr)
```

```
real(kind=8):: X0, X1  
  
call MPI_REDUCE  
(X0, X1, 1, MPI_DOUBLE_PRECISION, MPI_MAX, 0, <comm>, ierr)
```

```
real(kind=8):: X0(4), XMAX(4)  
  
call MPI_REDUCE  
(X0, XMAX, 4, MPI_DOUBLE_PRECISION, MPI_MAX, 0, <comm>, ierr)
```

Global Max values of X0[i] go to XMAX[i] on #0 process (i=0~3)

Example of MPI_Reduce (2/2)

```
call MPI_REDUCE  
(sendbuf, recvbuf, count, datatype, op, root, comm, ierr)
```

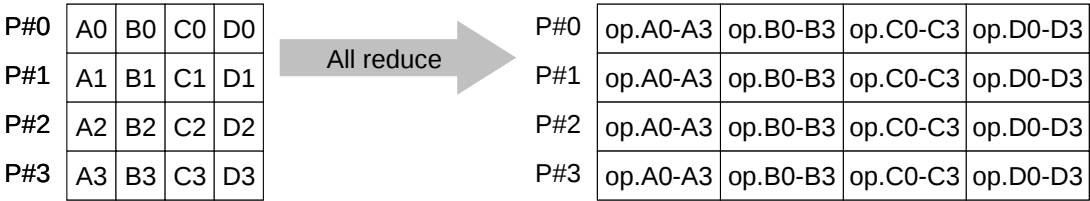
```
real(kind=8):: X0, XSUM  
  
call MPI_REDUCE  
(X0, XSUM, 1, MPI_DOUBLE_PRECISION, MPI_SUM, 0, <comm>, ierr)
```

Global summation of X0 goes to XSUM on #0 process.

```
real(kind=8):: X0(4)  
  
call MPI_REDUCE  
(X0(1), X0(3), 2, MPI_DOUBLE_PRECISION, MPI_SUM, 0, <comm>, ierr)
```

- Global summation of X0[0] goes to X0[2] on #0 process.
- Global summation of X0[1] goes to X0[3] on #0 process.

MPI_Allreduce



- MPI_Reduce + MPI_Bcast
- Summation (of dot products) and MAX/MIN values are likely to be utilized in each process
- call MPI_Allreduce
(sendbuf, recvbuf, count, datatype, op, comm)
 - sendbuf choice I starting address of send buffer
 - recvbuf choice 0 starting address receive buffer
type is defined by "datatype"
 - count int I number of elements in send/receive buffer
 - datatype MPI_Datatype I data type of elements of send/recive buffer
 - op MPI_Op I reduce operation
 - comm MPI_Comm I communicator

Program: 1d.c(11/11)

Output by Each Process

```
/*  
//-- OUTPUT  
*/  
printf("¥n¥s¥n", "### DISPLACEMENT");  
for (i=0; i<N; i++) {  
    printf("%3d%8d%16.6E¥n", MyRank, i+1, U[i]);  
}  
  
ierr = MPI_Finalize();  
return ierr;  
}
```

Output List: Not organized

### DISPLACEMENT		
0	1	0.000000E+00
0	2	1.000000E-01
0	3	2.000000E-01
0	4	3.000000E-01
0	5	4.000000E-01
0	6	5.000000E-01
0	7	6.000000E-01
0	8	7.000000E-01
0	9	8.000000E-01
0	10	9.000000E-01
0	11	1.000000E-00

### DISPLACEMENT		
1	1	1.100000E+00
1	2	1.200000E+00
1	3	1.300000E+00
1	4	1.400000E+00
1	5	1.500000E+00
1	6	1.600000E+00
1	7	1.700000E+00
1	8	1.800000E+00
1	9	1.900000E+00
1	10	2.000000E+00

### DISPLACEMENT		
3	1	3.100000E+00
3	2	3.200000E+00
3	3	3.300000E+00
3	4	3.400000E+00
3	5	3.500000E+00
3	6	3.600000E+00
3	7	3.700000E+00
3	8	3.800000E+00
3	9	3.900000E+00
3	10	4.000000E+00

### DISPLACEMENT		
2	1	2.100000E+00
2	2	2.200000E+00
2	3	2.300000E+00
2	4	2.400000E+00
2	5	2.500000E+00
2	6	2.600000E+00
2	7	2.700000E+00
2	8	2.800000E+00
2	9	2.900000E+00
2	10	3.000000E+00

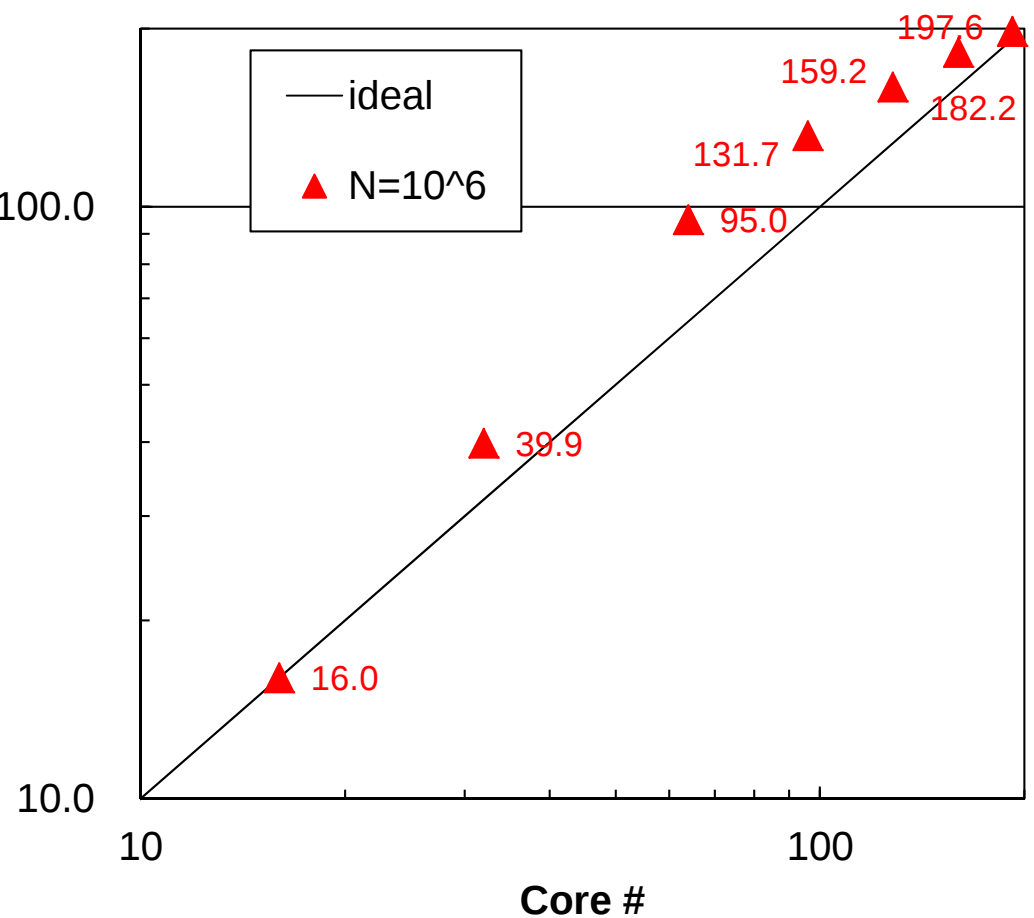
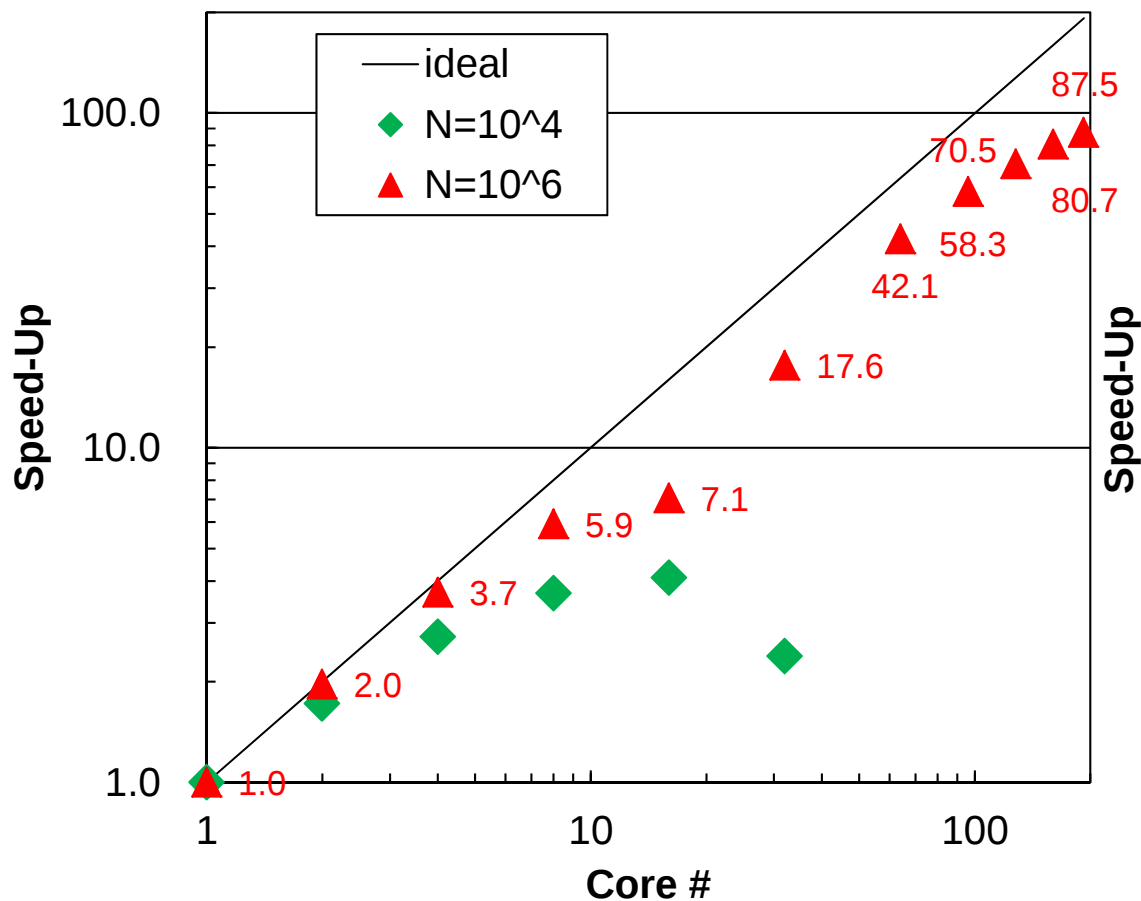
- Overview
- Distributed Local Data
- Program
- **Results**

Results for CG Solver

Time for 100 Iterations in $N=10^6$ case

based on
a core

based on
a node (16 cores)



Performance is lower than ideal one

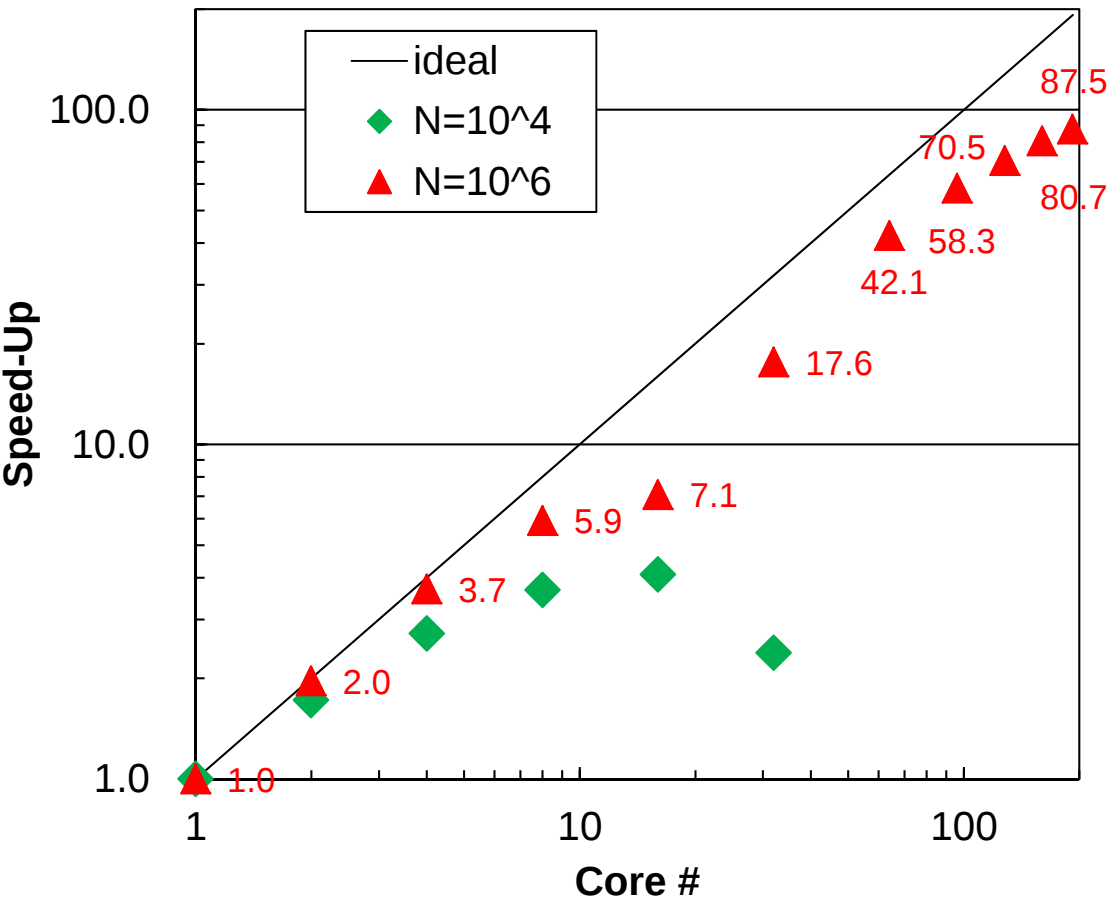
- Time for MPI communication
 - Time for sending data
 - Communication bandwidth between nodes
 - Time is proportional to size of sending/receiving buffers
- Time for starting MPI
 - latency
 - does not depend on size of buffers
 - depends on number of calling, increases according to process #
 - $O(10^0)$ - $O(10^1)$ μsec .
- Synchronization of MPI
 - Increases according to number of processes

Performance is lower than ideal one (cont.)

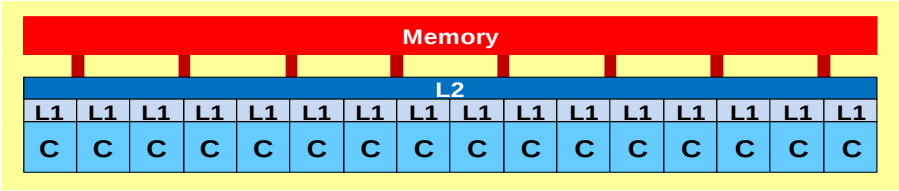
- If computation time is relatively small (N is small in S1-3), these effects are not negligible.
 - If the size of messages is small, effect of “latency” is significant.

Performance is not so good... between 1-16 cores

based on performance at a single core

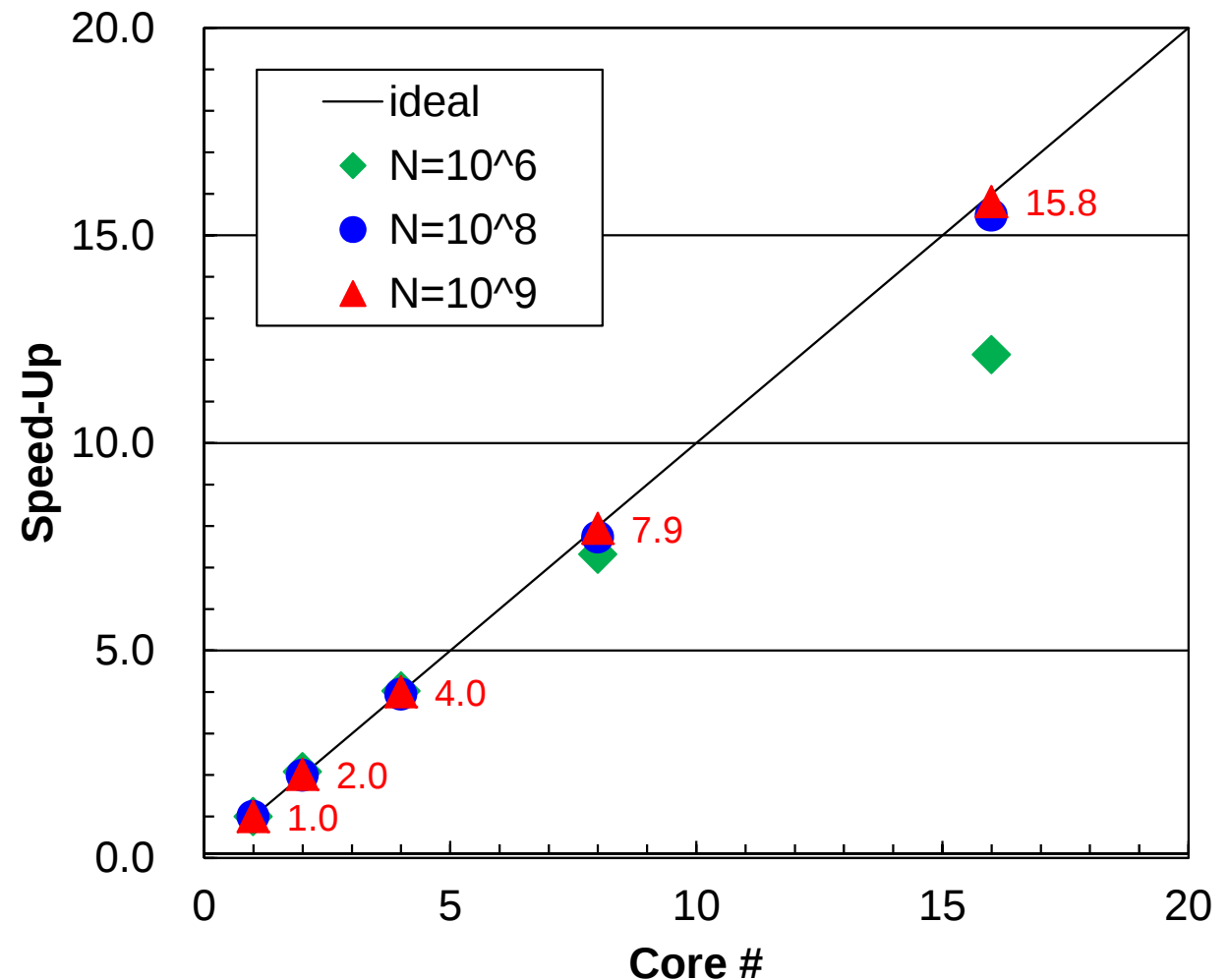


- x 7.1 at 16 cores, because of memory contention
 - STREAM
 - NOT mainly because of communication overhead

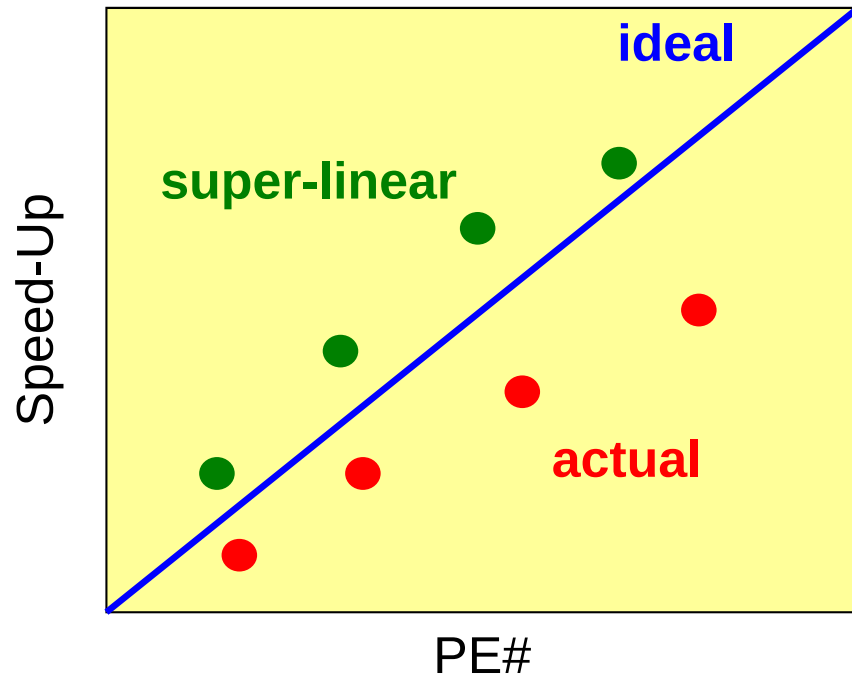


Not so significant in S1-3

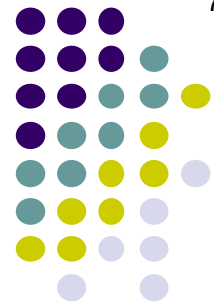
- ◆ : $N=10^6$, ● : 10^8 , ▲ : 10^9 , — : Ideal
- Based on performance at a single core (sec.)
- Trapezoidal rule: requirement for memory is very small (no arrays), NON memory-bound application



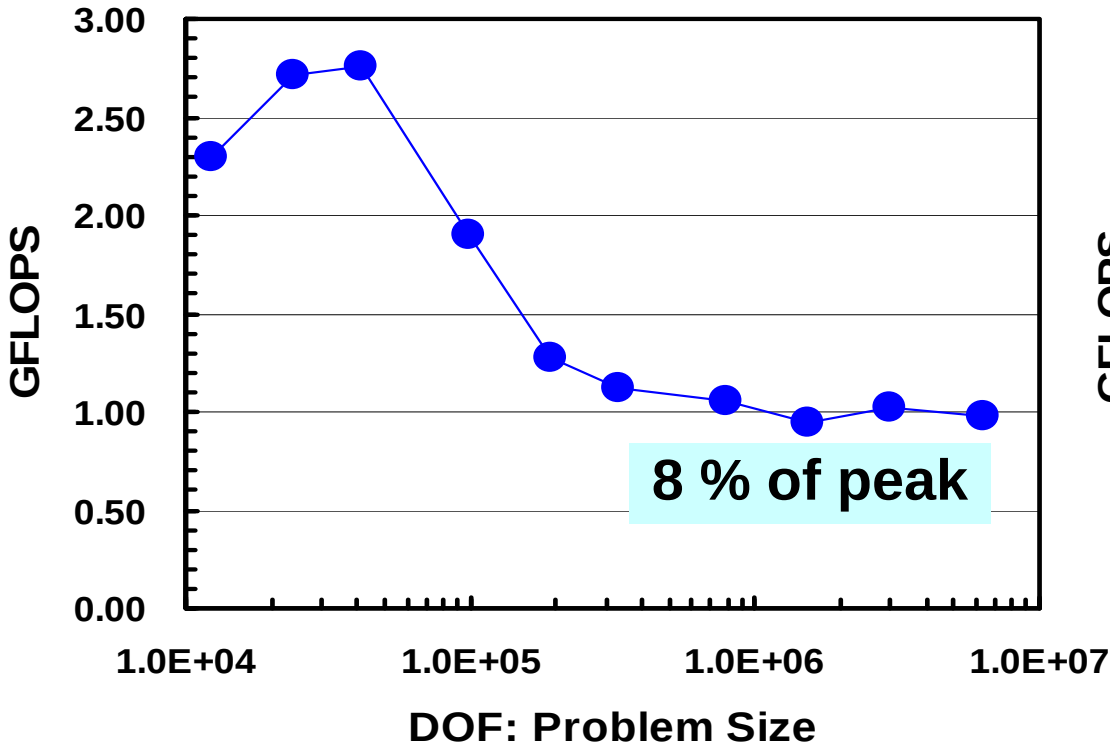
Super-Linear in Strong Scaling



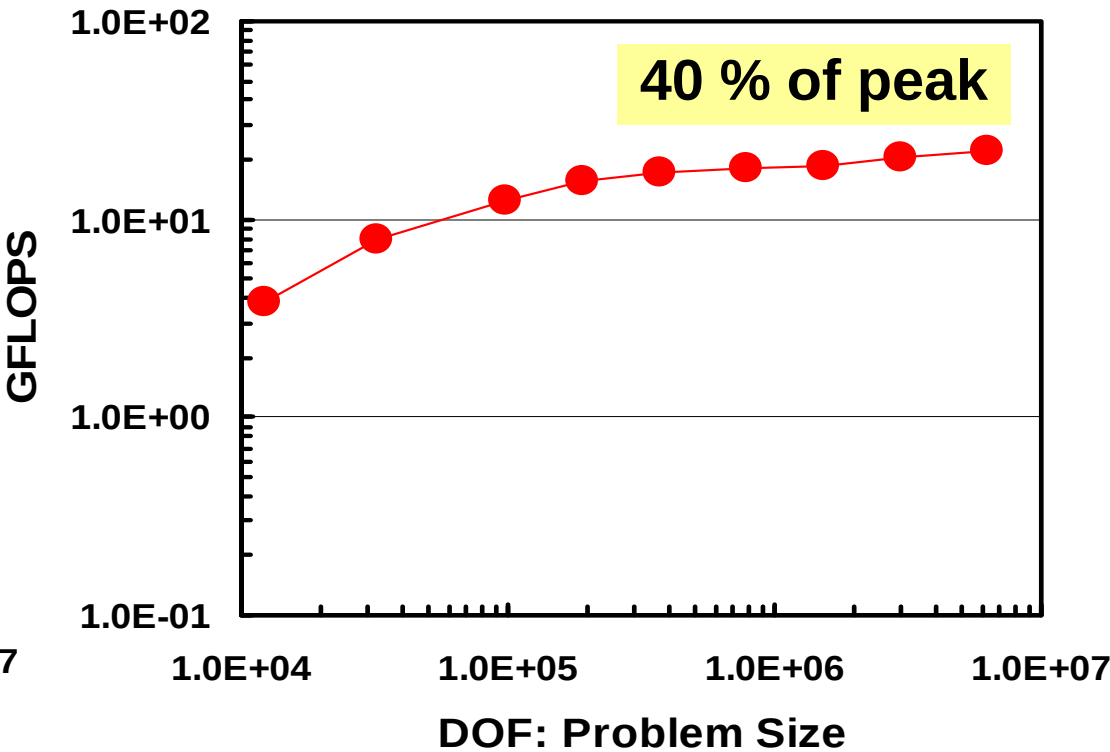
- In strong scaling case where entire problem size is fixed, performance is generally lower than the ideal one due to communication overhead.
- But sometime, actual performance may be better than the ideal one. This is called “super-linear”
 - only for scalar processors
 - does not happen in vector processors



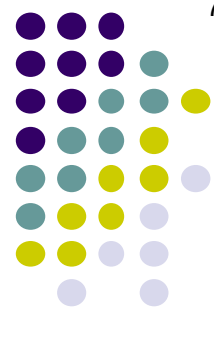
Typical Behaviors



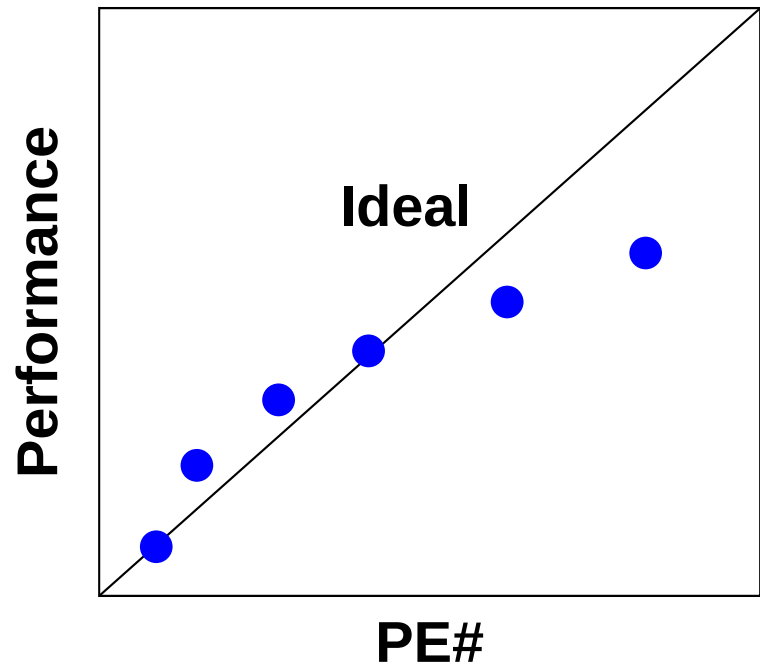
IBM-SP3:
Higher performance for small problems,
effect of cache



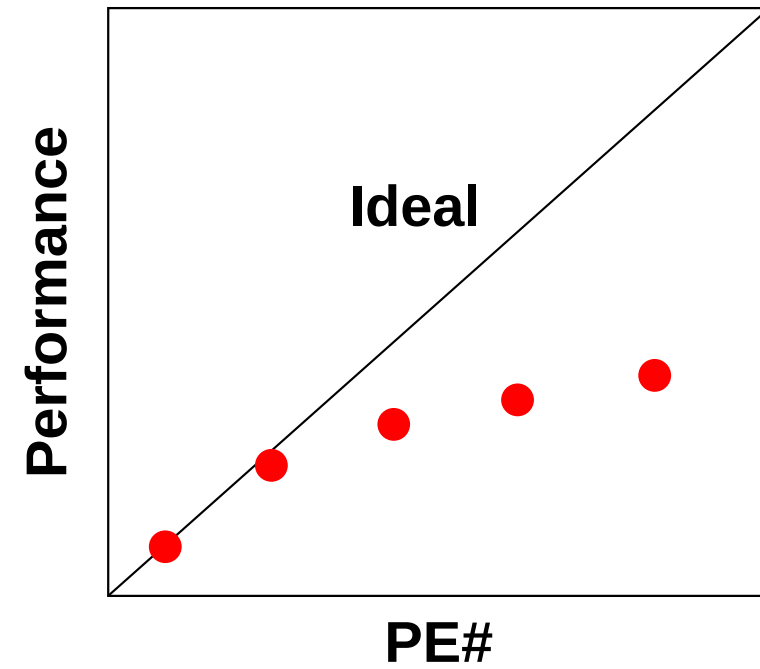
Earth Simulator:
Higher performance for large-scale
problems with longer loops



Strong Scaling



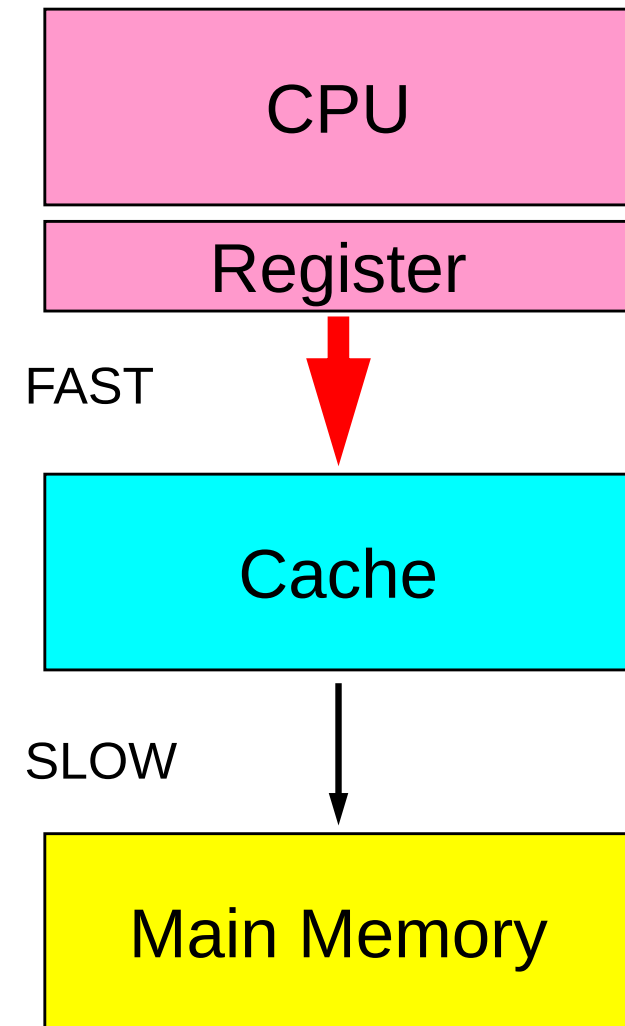
IBM-SP3:
“Super-linear” happens if number of PE is not so large. Performance is getting worse due to communication overhead if PE number is larger.



Earth Simulator:
Performance is getting worse due to communication overhead and smaller loop length if PE number is larger.

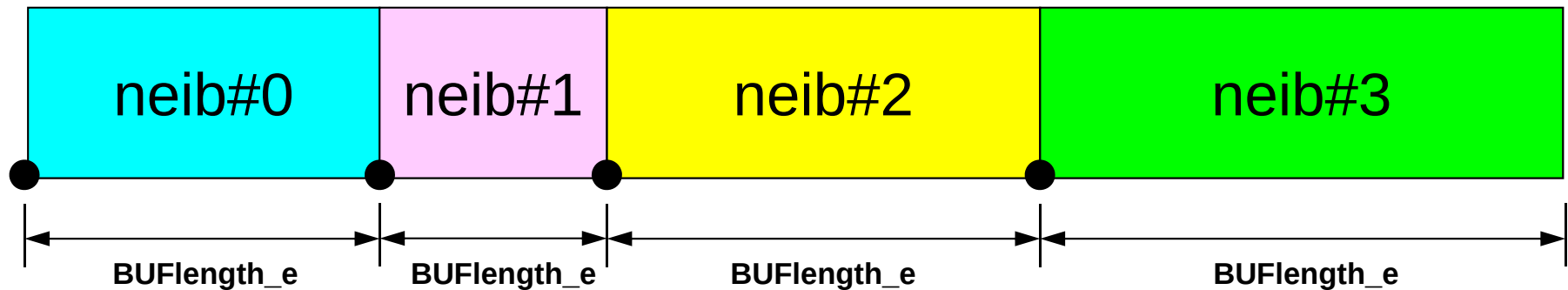
Why does “Super-Linear” happen ?

- Effect of Cache
- In scalar processors, performance for smaller problem is generally better.
 - Cache is well-utilized.



Memory Copy is expensive (1/2)

SendBuf



export_index[0] export_index[1] export_index[2] export_index[3] export_index[4]

export_item (export_index[neib]:export_index[neib+1]-1) are sent to neib-th neighbor

```
for (neib=0; neib<NeibPETot;neib++){
    for (k=export_index[neib];k<export_index[neib+1];k++){
        kk= export_item[k];
        SendBuf[k]= VAL[kk];
    }
}
```

Copied to sending buffers

```
for (neib=0; neib<NeibPETot; neib++){
    tag= 0;
    iS_e= export_index[neib];
    iE_e= export_index[neib+1];
    BUFlength_e= iE_e - iS_e

    ierr= MPI_Isend
        (&SendBuf[iS_e], BUFlength_e, MPI_DOUBLE, NeibPE[neib], 0,
         MPI_COMM_WORLD, &ReqSend[neib])
}

MPI_Waitall(NeibPETot, ReqSend, StatSend);
```

Memory Copy is expensive (1/2)

```
for (neib=0; neib<NeibPETot; neib++){
    tag= 0;
    iS_i= import_index[neib];
    iE_i= import_index[neib+1];
    BUFlength_i= iE_i - iS_i

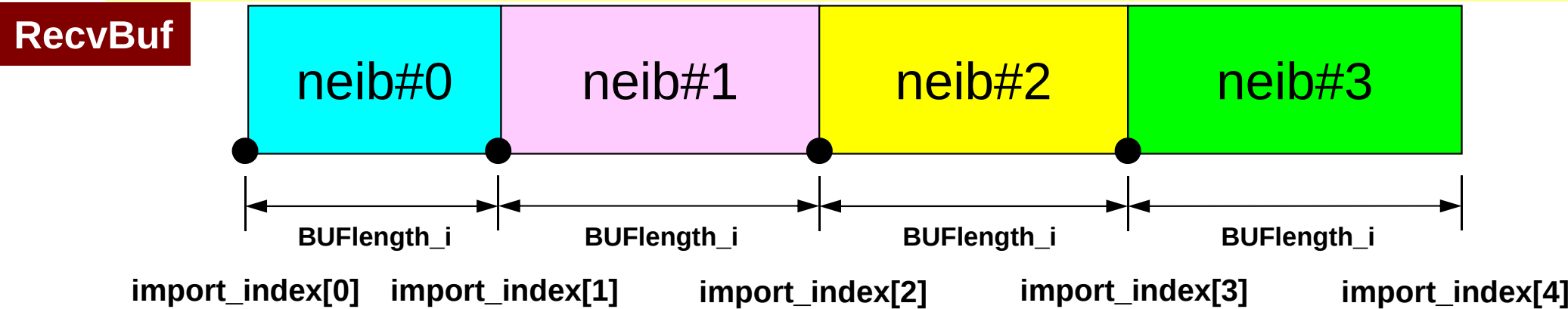
    ierr= MPI_Irecv
        (&RecvBuf[iS_i], BUFlength_i, MPI_DOUBLE, NeibPE[neib], 0,
         MPI_COMM_WORLD, &ReqRecv[neib])
}

MPI_Waitall(NeibPETot, ReqRecv, StatRecv);

for (neib=0; neib<NeibPETot;neib++){
    for (k=import_index[neib];k<import_index[neib+1];k++){
        kk= import_item[k];
        VAL[kk]= RecvBuf[k];
    }
}
```

Copied from receiving buffer

import_item (import_index[neib]:import_index[neib+1]-1) are received from neib-th neighbor



Summary: Parallel FEM

- Proper design of data structure of distributed local meshes.
- Open Technical Issues
 - Parallel Mesh Generation, Parallel Visualization
 - Parallel Preconditioner for Ill-Conditioned Problems
 - Large-Scale I/O

Distributed Local Data Structure for Parallel Computation

- Distributed local data structure for domain-to-domain communications has been introduced, which is appropriate for such applications with sparse coefficient matrices (e.g. FDM, FEM, FVM etc.).
 - SPMD
 - Local Numbering: Internal pts to External pts
 - Generalized communication table
- Everything is easy, if proper data structure is defined:
 - Values at boundary pts are copied into sending buffers
 - Send/Recv
 - Values at external pts are updated through receiving buffers